TAMPERE UNIVERSITY OF TECHNOLOGY

JIŘÍ HOFMAN

CODEBLOCK OPTIMIZATION OF JPEG-2000 BASED CODEC

Master of Science Thesis

Subject approved by the Department Council
January 19, 2005

Supervisors: Prof. Ioan Tăbuş
Ciprian Doru Giurcăneanu, Ph. D.

Preface

This Master of Science Thesis, **Codeblock Optimization of JPEG-2000 Based Codec**, was carried out in Institute of Signal Processing at Tampere University of Technology, Finland.

I would like to offer sincere thanks to my supervisors Prof. Ioan Tăbuş and Ciprian Doru Giurcăneanu, Ph. D., for their valuable ideas and patience.

I would also like to thank all developers of LaTeX which was used for writing this thesis. My gratitude also goes to Antti Larjo for his grammatical corrections. Finally, I would like to thank my family for their encouragement and support.

Tampere, on January 31, 2005

Jiří Hofman

Näyttelijänkatu 19 B 6
FI-33720 Tampere
FINLAND
tel. +358 504 661 860

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Institute of Signal Processing

JIŘÍ HOFMAN: Codeblock Optimization of JPEG-2000 Based Codec
Master of Science Thesis, 89 pages
Supervisors: Prof. Ioan Tăbuş and Ciprian Doru Giurcăneanu, Ph. D.
January 2005
Keywords: Image Compression, Codeblock optimization, JPEG-2000, EBCOT

The recently created JPEG-2000 standard brought new possibilities into image compression fields. The standard provides a complete normative description of a decoder but only informative description of an encoder. This enables to implement the encoder adapted to conditiones where it will be used. An important improvement of the encoded datastream can be achieved by optimization of the encoder.

In codeblock based compression of wavelet decomposition, the optimization of the entire image can be replaced by independent optimization of every codeblock. A succes of such a replacement is in a case of the JPEG-2000 style compression very high due to usage of an almost orthonormal wavelet transform.

This thesis describes an implementation of the JPEG-2000 based encoder and decoder and analyzes its advantages and drawbacks. The implementation comprises a kernel of the JPEG-2000 standard combined with EBCOT optimization method which produces the best possible code for each codeblock by overall optimization in the rate-distortion plane.

Contents

List of Figures

List of Tables

List of Algorithms

List of Used Symbols

$A \leftarrow B$	assigment; setting the value of the variable A
$A \gg B$	shift to the right (towards the least significant bit); A shifted by B bits to the right
$A \ll B$	shift to the left (towards the most significant bit); A shifted by B bits to the left
$A \wedge B$	logic *and*; A and B
$A \vee B$	logic *or*; A or B
$A \barwedge B$	bitwise logic *and*; A and B
$A \barvee B$	bitwise logic *or*; A or B
$0\mathrm{x}\,\mathrm{FA}\,96\,30$	hexadecimal number FA9630 (decadic 16422448)
$A \otimes B$	logic *xor*; A xor B
$\forall a$	for all a
$a \in B$	in, is a member of; a is a member of B
$\{\}$	set
$A{+}{+}$	increment; $A \leftarrow A + 1$
$\lceil A \rceil$	ceiling; ceiling of the variable A
$\lfloor A \rfloor$	floor; floor of the variable A
$\mathcal{B}$	set B
$\cup_{k=0}^{q} \mathcal{B}_k$	union of all sets $\mathcal{B}_k$ with indices from 0 to q inclusive
$A \subseteq B$	a set A is a subset of the set B; A can be also equal to B

Chapter 1

Introduction to the JPEG-2000 Standard

1.1 Purpose of the JPEG-2000

Nowadays world widely used image compression standard JPEG (*Joint Photographic Experts Group*) [1] suffers from several unpleasant problems. Firstly, the performance of codecs based on this standard is not relatively high anymore. Mainly so called *blocking-artefacts* cause a distortion which is very annoying. Blocking-artefacts appear in images encoded with high compression. Origin of this distortion is in division of the image into eight by eight pixels large blocks which are independently transformed by *discrete cosine transform* (DCT) and encoded. It is obvious that bigger differences between two blocks on their common border will appear with a lower number of bits used for description of each block. Apparently, a transform, which can be used for much larger areas, or even whole image, should be used. However, DCT does not provide as good results as we need in this case. That is why the JPEG consortium chose wavelet transform for the JPEG-2000 standard [3].

Besides the low encoding performance, a need of a unified standard for different purposes appeared with more and more extensive development of information technologies. The new JPEG-2000 standard provides both lossless (bit preserving) and lossy compression. These functionalities were previously split into two standards — JPEG and JPEG-LS [2]. Moreover, one can store all kinds of digital still images — bi-level, continuous-tone gray-scale, palletized colour and continuos-tone colour — in the JPEG-2000 file.

Also other features of the compressed data by the JPEG-2000 encoder are very appre-

ciated. Bitrate control is used to limit the bandwith needed to transfer the image. Coding of a whole image or at least of a whole tile at the same time, in opposite to processing block by block like in JPEG standard, provides a quality scalability which is helpful when a certain source image should be displayed by different devices in different resolutions. Small devices, like mobile phones, do not need to know all data of the image to display it in a low resolution. Another nice property of the JPEG-2000 is the possibility to specify a *region of interest* (ROI) which can be encoded in a higher quality. Of course, if the total bitrate for the image is kept, an improvement of a certain part of the image results in deterioration of the rest. However, the image can be much "nicer" to a human eye in this case. In many applications, another feature may be crucial — high error resilience. The stream syntax, which supports it, is defined in the JPEG-2000 standard as well.

1.2 General description of the JPEG-2000 Standard

First of all, it must be mentioned that the JPEG-2000 standard is mostly written from a point of view of the decoder. This guarantees that a stream, decribed by the standard, is always decodable but it also keeps huge freedom for implementations of the encoder. Simply said, the standard does not care how the encoded stream is obtained but only ensures that if the proper codestream was created, one can always decode it. Because the standard provides some information and hints how to implement the encoder, implementors must clearly distinct between normative and informative clauses in the standard.

The JPEG-2000 standard discusses all problems in 10 annexes.

- Codestream syntax
- Image and compressed image data ordering
- Arithmetic entropy coding
- Coefficient bit modelling
- Quantization
- Discrete wavelet transform of tile-components
- DC level shifting and multiple component transform
- Coding of images with regions of interest
- JP2 file format syntax
- Examples and guidelines

This list will be summarized in following sections. Yet before that, encoding principles of the JPEG-2000 will be described shortly. Encoder works in the following manner:

- Components of the image are divided into rectangular tiles.
- Wavelet transform on a tile-component, creating decomposition levels, is performed.
- The decomposition levels are made up of subbands of coefficients.
- The subbands of coefficients are quantized and collected into rectangular arrays of codeblocks.
- Each bitplane of the coefficients in the codeblock is entropy coded in three passes.
- If a region of interest (ROI) is defined, the coefficients to be encoded first will be those relevant for the ROI.

After this procedure the image data is converted to compressed image data stream. In order to get the final codestream, one must also:

- Collect the compressed image data from the coding passes in layers.
- Divide each layer into precints.
- Create packets from each precint.
- Interleave all packets from a tile in one of several orders and placing them in one, or more, tile-parts
- Create a main header at the beginning which describes the original image and various decompositions and coding styles.
- Create an optional file format which describes the meaning of the image and its components in the context of the application.

It is obvious, that the JPEG-2000 decoder must work in the opposite way in order to be able to recover the encoded image. Fig. 1.1 illustrates the principles of the decoder.

An important note must be mentioned too: following sections do not include the whole JPEG-2000 standard. It is not a goal of this thesis to rewrite it. Rather, the most interesting parts are described or discussed from a point of view of the standard or my implementation of the JPEG-2000 based encoder and decoder. Some parts will be described only from the encoder side in order to review briefly the standard and also to keep the issue understandable.

1.3 Codestream syntax

The JPEG-2000 standard precisely describes the syntax of the codestream which should be decodable. This codestream syntax description is not a file format.

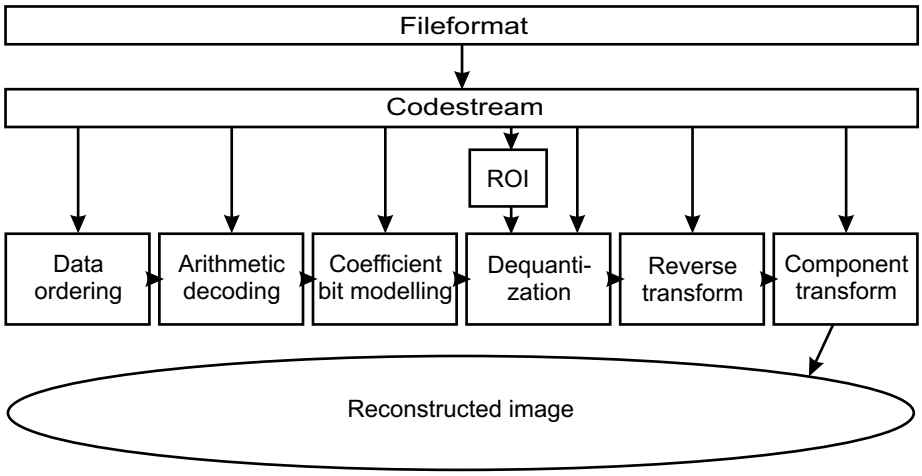

Figure 1.1: Block diagram of the JPEG-2000 decoder principles

The whole codestream consists of headers and bitstream. There are two kinds of headers: main and tile-part headers. They are collections of markers and marker segments. Some markers can appear in only one of two types of headers while others can be placed anywhere. Every marker segment includes a marker and associated parameters. The standard determines all possibly used markers and marker segments and their usage. Six types of markers and marker segments are used:

- *Delimiting* markers and marker segments to frame the main and tile-part headers and bitstream data. Each codestream has only one "start of codestream" marker, one "end of codestream" marker and at least one tile-part. Each tile-part must begin with one "start of tile-part" and contain one "start of data" marker indicating the beginning of bitstream data for the current tile-part.
- *Fixed informational* marker segments giving required information about the image. "Image and tile size" marker segment is required in the main header. It provides information about the uncompressed image such as width and height of the tiles, number of components and component bit depth.
- *Functional* marker segments to describe the coding functions used to code the entire tile or image depending on where the marker segment was found. For example number of decomposition levels, layering for compressing, region of interest or type of quantization.
- *In bitstream* markers and marker segments providing error resilience. They can be

found in the bitstream and denote the beginnings of packets and ends of packet headers within a codestream.

- *Pointer* marker segments providing specific offsets in the bitstream. These marker segments allow direct access into the bitstream because they provide pointers to tile-parts and packets.
- *Informational* marker segments giving subsidiary information. They are not necessary for decoder. Comments are one of the usage examples of these marker segments.

1.4 Ordering of image and compressed data

After achieving ability of codestream reading, implementor of the JPEG-2000 standard must understand to the ordering of image and compressed image data. The standard describes various structural entities and their organization in the codestream: components, tiles, subbands, precints and codeblocks.

An image is comprised of one or more *components*. Each component consists of a rectangular array of samples. Components can be sampled in different resolutions and have different sizes but all of them are mapped to the same *reference grid* in a specified way. Every *component sample* is associated with a reference grid point, however every reference grid point obviously need not be associated to any component sample. See example in fig. 1.2.

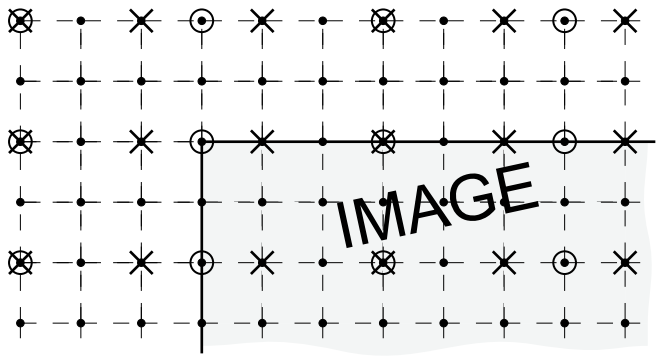

Figure 1.2: Upper left component sample locations.

Each component is divided into tiles according to tiling of the reference grid (see fig. 1.3). The tile-components are coded independently. Wavelet transform is used for transformation of each tile-component into several decomposition levels. Decompositions levels are related to *resolution levels*. The resolution levels consist of *subbands* — either HL, LH and HH, or

N_LLL. There is one more resolution level than there are decomposition levels because the lowest resolution level consists only of the LL subband of the lowest decomposition level.

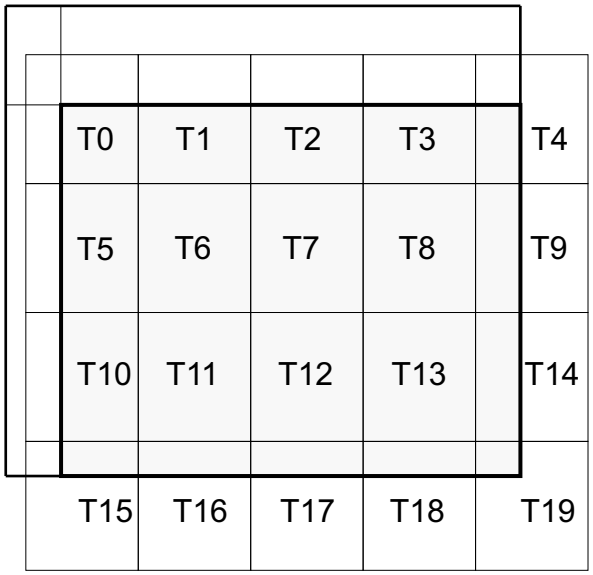

Figure 1.3: Tiling of the reference grid.

Partition of the subbands at the resolution level into *precints* is done similarly as the tiling of whole image. However, this division is not performed in order to divide components into smaller parts which are easier to handle, but to divide subbands into groups which will be saved as a bitstream altogether. Precincts are a collection of codeblocks (see below) which provide necessary data structures for this partition. This partition has no impact on the transform or coding of image samples. It serves only to organize codeblocks.

The subbands are divided into rectangular *codeblocks* for the purpose of coefficient modelling and coding. The codeblock size is the same for all resolution levels but it is also bounded by the precinct size. Codeblocks may extend beyond the boundaries of the subband coefficient. However, if this happens, only coefficients lying within the subband are coded.

Codeblocks are processed consecutively, bitplane by bitplane, coding pass by coding pass (see section 1.6). The coding passes are grouped into *layers*, also known as layers of quality. Each layer contains some number of coding passes from each codeblock in the tile. The number of coding passes may be different for different codeblocks and can be also zero.

Packets are segments of compressed image data in the final bitstream representing a

specific tile, layer, component, resolution level and precinct. Packet data is aligned at 8-bit boundaries. Because each resolution level contains either N_LLL or HL, LH and HH band, the compressed image data in a packet contains either only N_LLL or HL, LH and HH band in this order (see section 1.8).

1.5 Arithmetic entropy coding

Arithmetic entropy coding process which is fully normatively defined in the JPEG-2000 standard is based on a recursive probability interval subdivision of *Elias coding*. The input of the encoder is a pair CX and D — context and decision. The output is compressed image data. With each binary decision the current probability interval is subdivided into two subintervals and the code string is modified so that it points to the lower bound of the subinterval corresponding to the *symbol* which occurred. In fact, more (MPS) and less (LPS) probable symbols are coded rather than zeros and ones. When the more probable symbol should be coded, the interval assigned to the less probable symbol is added to the code string. Since the coding process involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can often be coded at a cost of much less than one bit per decision.

Fractional values are represented by integer numbers. Precisely, 0x 80 00 is equivalent to 0.75. The interval is kept in the range $0.75 \leq \mathrm{A} < 1.5$ [1]. It is doubled whenever the integer value falls below 0x 80 00. Similarly register C is also doubled when A is doubled.

Since encoding process is more synoptic for understanding the principles of coding than decoder, the arithmetic encoding procedure of the JPEG-2000 standard is described more deeply in a couple of following algorithms. However, it is only informative in the standard. Meanings of used variables can be found in table 1.1.

The whole algorithm is composed of three parts — initialization, encoding in the loop and flush of the data. The first two parts are directly written in alg. 1.1. The flush is called from it and can be found in alg. 1.7.

Firstly, the encoder must be initialized. It includes settings for MPS and I arrays, A and C registers and pointer to B (BP). MPS(i) and I(i) initial values are taken from the table of inital values. The A register is set to zero and shift counter set to 12. That means there are three spacer bits in the register which need to be filled before the field from which

[1] 1.5 is equivalent to 0x 1 00 00

D	decision
CX	context
A-register	interval value register
C-register	code register
B	compressed image data buffer
BP	pointer to B
BPST	pointer to the first byte of compressed image data
CT	shift counter
I[CX]	state of the finite-state machine
Qe[i]	Qe value in the state i
NMPS[i]	next state for MPS renormalization in state i
NLPS[i]	next state for LPS renormalization in state i

Table 1.1: List of symbols in the JPEG-2000 arithmetic encoder

the bytes are removed is reached. If the compressed image data buffer is a 0x FF byte, CT must be increased in order to compensate a spurious bit stuff which occurres due to it.

Algorithm 1.1 Encoder

```
A ← 0x 80 00; C ← 0  {Initialize encoder}
INIT MPS[i] and I[i] for all contexts i
BP ← BPST − 1; CT ← 12
if B = 0x FF then
  CT = 13
end if {End of initializing}
repeat
  Read CX,D
  call ENCODE {alg. 1.2}
until Encoding is finished
call FLUSH {alg. 1.7}
```

Then the encoder begins to do its job — it reads decisions and contexts and encodes them in a loop. When all decisions are encoded, the Flush procedure is executed (see page 11).

Encoding of decision (zeros and ones) in the loop is done in one of two ways, either

CodeLPS or CodeMPS. The algorithm decides between them according to input decision D and MPS of the context CX (see alg. 1.2).

Algorithm 1.2 Encode

```
if D = 0 then
  if MPS(CX) = 0 then
    call CODEMPS {alg. 1.4}
  else
    call CODELPS {alg. 1.3}
  end if
else
  if MPS(CX) = 1 then
    call CODEMPS {alg. 1.4}
  else
    call CODELPS {alg. 1.3}
  end if
end if
```

Encoding an MPS (see alg. 1.4) and LPS (see alg. 1.3) basically consist of a scaling of the interval to Qe(I(CX)), where I(CX) is the index to the given table (for details see [3, annex C]) and Qe(I(CX)) is the estimated probability for this index. Besides the probability estimation, the table contains NLPS and NMPS — indices which will be used next time. So that, the probability estimation can be considered as a finite-state machine. The table also contains the SWITCH flag for each index I(CX). If it is set, then MPS(CX) is inverted. This feature saves some bits from the resulting encoded bitstream.

In both of the procedures 1.3 and 1.4, the upper interval is first calculated and compared to the lower one in order to confirm that Qe has smaller size. If not, the code register is updated, Qe is added to it and the procedures are finished by renormalization.

Renormalization procedure (see alg. 1.5) generates the outputting bits. The A and C registers are shifted bit by bit. The number of shifts is counted in the counter CT. When CT reaches zero, a byte of compressed data is removed from C by the routine ByteOut.

The ByteOut procedure (see alg. 1.6) outputs a content of the C register to the output bitstream, on the position where BP is pointing to. Bit stuffing occurres in this routine. It is needed to limit the carry propagation into the completed bytes of compressed image

Algorithm 1.3 CodeLPS — Encode Less Probable Symbol

```
A ← A − Qe(I(CX))
if A < Qe(I(CX)) then
  C ← C + Qe(I(CX))
else
  A ← Qe(I(CX))
end if
if SWITCH(I(CX)) = 1 then
  MPS(CX) ← 1 − MPS(CX)
end if
I(CX) ← NLPS(I(CX))
call ENCRENORM {alg. 1.5}
```

Algorithm 1.4 CodeMPS — Encode More Probable Symbol

```
A ← A − Qe(I(CX))
if A = A − Qe(I(CX)) then
  if A < Qe(I(CX)) then
    A ← Qe(I(CX))
  else
    C ← C + Qe(I(CX))
  end if
  I(CX) = NMPS(I(CX))
  call ENCRENORM {alg. 1.5}
else
  C ← C + Qe(I(CX))
end if
```

Algorithm 1.5 EncRenorm — Encoding Renormalization

```
do
  A ← A ≪ 1; C ← C ≪ 1
  CT ← CT − 1
  if CT = 0 then
    call BYTEOUT {alg. 1.6}
  end if
while A ⩓ 0x 80 00 = 0
```

data. The conventions used make it impossible for a carry to propagate through more than the byte most recently written to the compressed image data buffer.

Algorithm 1.6 ByteOut — Byte Output

```
if B ← 0x FF then
  BP ← BP + 1; B ← C ≫ 20
  C ← C ⩓ 0x F FF FF; CT ← 7 {bit stuffing}
else
  if C < 0x 8 00 00 00 then
    BP ← BP + 1; B ← C ≫ 19
    C ← C ⩓ 0x 7 FF FF; CT ← 8
  else
    B ← B + 1
    if B = 0x FF then
      C ← C ⩓ 0x 7 FF FF FF
      BP ← BP + 1; B ← C ≫ 20
      C ← C ⩓ 0x F FF FF; CT ← 7 {bit stuffing}
    else
      BP ← BP + 1; B ← C ≫ 19
      C ← C ⩓ 0x 7 FF FF; CT ← 8
    end if
  end if
end if
```

The Flush algorithm (see alg. 1.7) terminates the encoding operations and generates the required terminating marker. The first part of the Flush procedure sets as many bits in the C register to 1 as possible. Upper bound for C register — sum of the code and interval registers — is found. Lower 16 bits of C register are forced to 1 and the result is compared to the upper bound. If C is too big, the code register C will be reduced to a value which is within the interval. Later, all data from the code register is output and 0x FF byte is placed at the final bits of the compressed image data. The procedure guarantees that any marker code at the end of the compressed image data will be recognized before decoding is complete.

An arithmetic decoder works almost in the same way as the encoder does. Of course, the main difference is that encoder produces bits and the decoder consumes them. However,

Algorithm 1.7 Flush

```
TEMPC ← C + A; C ← C Ⓥ 0x FF FF
if C ≥ TEMPC then
  C ← C − 0x 80 00
end if
C ← C ≪ CT
call BYTEOUT {alg. 1.6}
C ← C ≪ CT
call BYTEOUT {alg. 1.6}
if B ← 0x FF then
  Discard B
else
  BP ← BP + 1
end if
```

choosing between LPS and MPS branch of the algorithm, moving among states of the finite-state machine and renormalization function similarly.

Because the encoder is usually more illuminating for understanding the principles of arithmetic coding, the detailed description of the JPEG-2000 arithmetic decoder will be skipped.

1.6 Coefficient bit modelling

Processing of each codeblock (see section 1.4) will be discussed in this section. Only encoder will be described here in order to keep interpretation understandable.

Codeblocks are coded bitplane by bitplane from the most significant bitplane with a nonzero bit to the least significant bitplane. The bitplanes are scanned in a special order — the codeblock is divided into four samples high *belts* and each belt is scanned in four bytes long vertical *stripes* from left to right. The belts are scanned from top to bottom of the codeblock. If the codeblock height is not divisible by four, the last belt will have less than four samples in each stripe.

Each bitplane is scanned actually three times because the whole coding is performed in three passes: *significance propagation*, *magnitude refinement* and *cleanup* pass.

Each coefficient in the codeblock has an associated binary state variable called its *sig-*

nificance state. The coefficients are *insignificant* in the beginning of the coding process and become *significant* at the bitplane where the most significant magnitude bit equal to 1 is found. The coefficient's context vector is a binary vector consisting of the significance states of its 8-nearest neighbour coefficients. The nearest neighbours lying out of the codeblock are considered as insigificant.

Not every combination of significant states of the nearest neighbours creates an independent context vector. The context vectors are grouped into only 17 *contexts*. Context formation rules are defined for each of the coding passes and sign coding. The contexts (or *context labels*) are provided to arithmetic encoder or decoder.

The first bitplane of the current codeblock with a non-zero element is processed only by cleanup pass. The remaining bitplanes are coded in three passes. Each coefficient bit is coded in exactly one of them. Which pass processes a certain coefficient bit, depends on conditions for that pass. Briefly, the significance propagation pass processes the coefficients that are predicted to become significant. The magnitude refinement pass includes bits from already significant samples and the cleanup pass processes all remaining bits. The sign bits are processed immediately after the appropriate samples become significant.

1.6.1 Significance propagation pass

A *context vector* of each sample is created from eight surrounding neighbour coefficients and mapped into one of nine contexts. Significant coefficients give a value 1 and insignificant coefficients give a value 0 for the creation of the context. Fig. 1.4 shows how surrounding samples are used for the context vector and context determination. All kinds of neighbours are processed in the same way — horizontal, vertical and diagonal *contributions* are summed. The contexts are determined from these summations. The mapping into contexts also depends on the subband. Table 1.2 contains rules for the context label determination.

The significance propagation pass includes only bits of coefficients whose significance states have not yet been set and have a non-zero context (see alg. 1.8). All other coefficients are skipped. The context is sent to the arithmetic encoder. If the value of the bit is 1, then the significance state is set to 1 and the immediate next bit to be encoded is the sign for the coefficient. Otherwise, the significance state remains 0. If the significance state of this sample is needed for computing the context vector of any following coefficient, the newest significance state for this sample will be used.

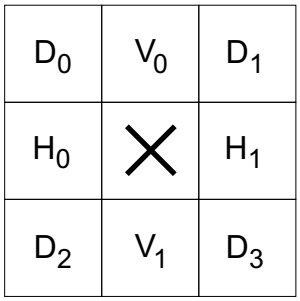

Figure 1.4: Neighbours states forming the context vector

Algorithm 1.8 Significance propagation pass

for all coefficients in the codeblock **do**
  compute context of the current sample
  if sample is insignificant and has a non-zero context **then**
    send the bit and context {**to arithmetic encoder**}
    if bit is 1 **then**
      call ENCODE_SIGN to encode sign of the coefficient {alg. 1.9}
    end if
  end if
end for

1.6.2 Sign bit coding

Another context vector is used to determine the context label for sign bit coding. The diagonal neighbours are not interesting anymore. The whole context label determination is done in two steps. In the first one, the contributions from vertical and horizontal neighbours are computed (see table 1.3). Each neighbour can be in one of three states: significant positive (S+),significant negative (S−) or insignificant (I).

The second step reduces nine posible horizontal and vertical configurations to one of five context labels (see table 1.4). One should notice that the context labels are again indexed only for the identification convenience. Besides the context determination, the *XORbit* is determined in this step as well. The XORbit is used similarly in the encoder and decoder. The encoder wants to save the sign bit of the current coefficient (a 1 bit means a negative coefficient and a 0 indicates a positive coefficient). This bit is XORed ($\otimes$, logical exclusive

LL[a] and LH			HL			HH		cntx[b]
$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum R_i^c$	$\sum D_i$	
2	–[d]	–	–	2	–	–	≥ 3	8
1	≥ 1	–	≥ 1	1	–	≥ 1	2	7
1	0	≥ 1	0	1	≥ 1	0	2	6
1	0	0	0	1	0	≥ 2	1	5
0	2	–	2	0	–	1	1	4
0	1	–	1	0	–	0	1	3
0	0	≥ 2	0	0	≥ 2	≥ 2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

[a] LL, LH, HL and HH are subbands

[b] *cntx* means the context label, it is only a identification convenience

[c] $\sum R_i = \sum(H_i + V_i)$

[d] – means "do not care"

Table 1.2: Contexts for significance propagation and cleanup passes

		V_0 or H_0		
		S+	I	S−
V_1	S+	1	1	0
or	I	1	0	−1
H_1	S−	0	−1	−1

Table 1.3: Contributions of the neighbours to the sign context

OR) with the XORbit

$$D = signbit \otimes \text{XORbit}. \tag{1.1}$$

The result is saved. See alg. 1.9.

On the other hand, the decoder wants to read the coming data. Because of a property of the XOR operation

$$\forall a, b; a, b \in \{true, false\} : (a \otimes b) \otimes b = a, \tag{1.2}$$

decoder also performs the XOR operation on the coming data and XORbit

$$signbit = D \otimes \text{XORbit}. \tag{1.3}$$

		Vertical		
		-1	0	1
Horizontal	-1	13/1	12/1	11/1
	0	10/1	9/0	10/0
	1	11/0	12/0	13/0

Table 1.4: Sign contexts / XORbits from the contributions

Algorithm 1.9 Sign bit encoding

compute horizontal contribution
compute vertical contribution
compute context label
compute XORbit
$D \leftarrow signbit \otimes$ XORbit
send D to arithmetic encoder

1.6.3 Magnitude refinement pass

The bits from coefficients that are already significant are processed in the magnitude refinement pass. Of course, the bits which have just become significant in the last significance propagation pass are not included. The context is determined by the summation of the current significance states of all eight neighbours and by whether this is the first bit of the coefficient to refine or not. Table 1.5 shows three contexts for this pass. The contexts are sent to the arithmetic coder along with the appropriate bits.

$\sum(H_i+V_i+D_i)$	First refinement	context
$-^a$	no	16
≥ 1	yes	15
0	yes	14

[a] – means "do not care"

Table 1.5: Contributions of the neighbours to the sign context

Algorithm 1.10 Magnitude refinement pass

```
for all previously significant coefficients in the codeblock do
  if this is the first refinement of the coefficient then
    sum the significnt states of the neighbours
    if sum = 0 then
      send the bit and context 14 {to arithmetic encoder}
    else
      send the bit and context 15
    end if
  else
    send the bit and context 16
  end if
end for
```

1.6.4 Cleanup pass

Cleanup pass handles coefficients which were previously insignificant and not processed by the last significance propagation pass. The cleanup pass does not only use the neighbour context from table 1.2 but also a *run-length* context.

The neighbour contexts for the coefficients are recreated in this pass using table 1.2. The coefficients which were found to be significant in the significance propagation pass are considered to be significant now.

The run-length context is a unique context. It is used when all four contiguous coefficients in the stripe are decoded in the cleanup pass and the context label for all of them is 0. For detailed description on how the runlength works, the encoder case will be used again.

If all four coefficients in the stripe, which should be encoded in the run-length, are insignificant, then symbol 0 is sent to arithmetic encoder with the run-length context. Otherwise symbol 1 is sent with the run-length context and the procedure continues. At least one of the coefficients in the stripe is significant. The next two bits (most significant bit first) sent with a special *uniform* context denote which coefficient in the stripe is the first significant one. The sign of that coefficient is encoded as it was described in subsection 1.6.2. Encoding of remaining coefficients continues in the manner described in subsection 1.6.1.

In case that four coefficients in the stripe are not coded in the cleanup pass or the context of any of them is non-zero, then the coefficients are again coded exactly as it was described in subsection 1.6.1.

Algorithm 1.11 Cleanup pass

```
for all complete stripes in the codeblock do
  compute contexts of all coefficients in the stripe
  if all four coefficients should be encoded in this pass then
    if all four coefficients are insignificant and their contexts are zero then
      send 0 and run-length context {to arithmetic encoder}
    else
      send 1 and run-length context
      send position of the 1st signif. coef. and uniform context
      call ENCODE_SIGN to encode sign of the 1st signif. coef. {alg. 1.9}
      for all remaining coefficients in the stripe do
        encode the bit of the coefficient
        if bit is 1 then
          call ENCODE_SIGN to encode sign of the coefficient {alg. 1.9}
        end if
      end for
    end if
  else
    for all coefficients in the stripe which should be processed in the cleanup pass do
      encode significance of the coefficient
      if bit is 1 then
        call ENCODE_SIGN to encode sign of the coefficient {alg. 1.9}
      end if
    end for
  end if
end for
call Cleanup pass — continuing {alg. 1.12}
```

Algorithm 1.12 Cleanup pass — continuing

```
for all remaining incomplete stripes in the codeblock do
  for all coefficients in the stripe do
    encode significance of the coefficient
    if bit is 1 then
      call ENCODE_SIGN to encode sign of the coefficient {alg. 1.9}
    end if
  end for
end for
```

An overview of the encoder side cleanup pass is in alg. 1.11 and 1.12.

1.6.5 Additional notes

The JPEG-2000 standard discusses also a couple of other detail issues in the annex of coefficient bit modelling. Notable problems are mainly *initializing* and *reinitializing* of the contexts, termination of the codestream, *error resilience* segmentation symbol, *vertically casual context* formation and *selective arithmetic coding bypass*. All these features place special markers or marker segments into the outputting bitstream.

When the contexts are *reinitialiazed*, they are set to the starting values which are specified by the standard. The arithmetic encoder can be terminated either at the end of every coding pass or only at the end of every codeblock.

Error resilience segmentation symbol ensures that error in the decoding of each bitplane can be detected. It is placed at the end of each bitplane. The correct decoding of this symbol confirms correctness of the decoding of this bitplane.

The *vertically causal context* formation allows ignoring the significance states of the samples in the stripes which are above the belt where currently processed coefficient is located. The significance states of the ignored coefficients are considered to be 0.

The *selective arithmetic coding bypass* enables usage of the arithmetic encoding only for the first ten passes (four bitplanes) and the remaining bitplanes are passed either from input to decoder, or from encoder to output skipping the arithmetic encoder. The raw (or lazy) mode can be used only for some of the remaining bitplanes. It can be terminated by a special terminating flag and arithmetic encoder can be restarted again. This can be repeated for every single coding pass.

1.7 Quantization

Quantization can be done according to the JPEG-2000 standard in two ways: *reversibly* and *irreversibly*. The reversible quantization means that quantization is not performed — the *quantization step* $\triangle_b$ is equal to 1. It is used when quantization should not cause any distortion to compressed image — after a reversible wavelet decomposition. The reversible quantization is not described more deeply in this thesis because implementation described in chapter 3 uses only the irreversible quantization.

The irreversible quantization is apparently used when the wavelet decomposition has already brought a distortion. In this case, a special kind of quantization can actually help to suppress its unintentional side effects.

The standard defines only inverse quantization procedure (dequantization in the decoder). The forward quantization procedure in the encoder is mentioned only informatively. However, for better understanding of the quantization, it will be described here later too.

Each wavelet transform coefficient (u, v) of the subband b has a value

$$\bar{q}(u,v) = (1 - 2s(u,v)) \sum_{i=1}^{N_b(u,v)} \left(2^{M_b - i} MSB_i(b,u,v)\right), \tag{1.4}$$

where $s(u, v)$ is a sign bit of the coefficient (1 for negative, 0 for positive), $N_b(u, v)$ is the number of decoded bits, $MSB_i(b, u, v)$ is the i^{th} bit of the coefficient (starting with the most significant bit) and M_b is given by

$$M_b = G + \varepsilon_b - 1, \tag{1.5}$$

where G is the number of *guard bits* and ε_b is an exponent of the quantization step $\triangle_b$. Guard bits are additional most significant bits added to each sample in order to prevent possible overflow. G, ε_b and $N_b(u, v)$ are specified in the appropriate marker segments.

The quantization step $\triangle_b$ for a given subband b is defined as

$$\triangle_b = 2^{R_I \log_2 gain_b - \varepsilon_b} \left(1 + \frac{\mu_b}{2^{11}}\right), \tag{1.6}$$

where R_I is the number of bits used to represent the original tile-component samples, $gain_b$ is the *subband gain* of the current subband b (see table 1.6 — a higher subband gain means a higher quantization step), ε_b is the exponent and μ_b is the mantissa. The denominator 2^{11} is due to the allocation of 11 bits in the codestream for μ_b. The mantissa is specified similarly to the exponent in the appropriate marker segments in the bitstream. The exponent and

mantissa pairs (ε_b, μ_b) are signaled in the codestream either for every subband or only for the N_LLL subband and derived for all other subbands. In the latter case, the derivation is

$$(\varepsilon_b, \mu_b) = (\varepsilon_{LL} - N_{LL} + n_b, \mu_{LL}), \tag{1.7}$$

where n_b denotes the number of decomposition levels from the original tile-component to the subband b.

subband b	$\log_2 gain_b$
LL	0
LH	1
HL	1
HH	2

Table 1.6: Subband gains

Reconstruction of the transform coefficient is simple

$$Rq(u,v) = \begin{cases} (\bar{q}(u,v) + r2^{G+\varepsilon_b-1-N_b(u,v)})\ \triangle_b & for \quad \bar{q}(u,v) > 0 \\ (\bar{q}(u,v) - r2^{G+\varepsilon_b-1-N_b(u,v)})\ \triangle_b & for \quad \bar{q}(u,v) < 0 \\ 0 & for \quad \bar{q}(u,v) = 0 \end{cases} \tag{1.8}$$

where r is a *reconstruction parameter* which can be arbitrarily chosen by the decoder for example to produce the best quality for the reconstruction. A common value is $r = 0.5$ but can be anything in the range $0 \leq r < 1$.

The forward quantization is done after forward wavelet transform. All transform coefficients $a(u,v)$ of the subband b are quantized to the value $q(u,v)$:

$$q(u,v) = sign(a(u,v)) \left\lfloor \frac{|a(u,v)|}{\triangle_b} \right\rfloor \tag{1.9}$$

The exponent ε_b and the mantissa μ_b corresponding to $\triangle_b$ can be derived from equation (1.7) and must be recorded in the codestream in the appropriate markers.

1.8 Discrete wavelet transform of tile-components

In this section, similarly to previous ones, will be described again mainly encoder side of the discrete wavelet transform which is called the *forward* discrete wavelet transform

(FDWT). Transformed signal can be perfectly or nearly perfectly reconstructed using an *inverse* discrete wavelet transform (IDWT). The perfect reconstruction is possible when the 5-3 reversible wavelet transform is used. And on the other hand, the nearly perfect reconstruction can be gained when the 9-7 irreversible wavelet transform is used to encode the image. Although the reversible wavelet transform seems to be advantageous, the irreversible wavelet transform produces significantly better results in lossy compression. The reversible wavelet transform can theoretically be used for lossy compression as well. However, it is not used in that way due to its worse results. The reversible wavelet transform is used when either lossless compression is needed or an arithmetic unit, which should perform all the computations, can calculate only in integer domain.

As was mentioned before, tile-components of the entire image are transformed independently. Then, it is important to mention how the FDWT works in general. The FDWT described in the JPEG-2000 standard uses a one-dimensional *subband decomposition* of a one-dimensional array of samples into *low-pass* coefficients, representing a downsampled residual version of the original array, and into *high-pass* coefficients, representing a downsampled residual version of the original array, needed to reconstruct the original array from the low-pass array.

The FDWT transforms DC-level shifted tile-component samples into a *set of subbands*. Necessary input of the FDWT procedure is also the number of decomposition levels N_L, which must be saved to the bitstream in appropriate markers.

The subbands are labelled by an index lev, corresponding to the decomposition level, followed by a signature of the filter type which is used to produce it. Possible variations are LL, HL, LH and HH. For example, the subband $b = lev\text{LH}$ corresponds to a downsampled version of $(lev - 1)$LL which has been high-pass filtered vertically and low-pass filtered horizontally. The subband 0LL is identical to the original tile-component.

The FDWT procedure (see alg. 1.13) is recursive and produces usual dyadic decomposition structure [11]. The whole tile-component is taken as the zeroth LL subband. The recursive part of the algorithm is repeated N_L times. In each iteration, the LL subband of the current decomposion level lev is decomposed by 2D_SD procedure. The total number of coefficients is not changing during iterations.

The 2D_SD procedure performs a decomposition (see fig. 1.5) of a two-dimensional array of coefficients or samples $a_{(lev-1)LL}(u, v)$ into four groups of subband coefficients $a_{levLL}(u, v)$, $a_{levHL}(u, v)$, $a_{levLH}(u, v)$ and $a_{levHH}(u, v)$.

The whole 2D_SD procedure consists of three steps (see alg. 1.14). In the first two steps,

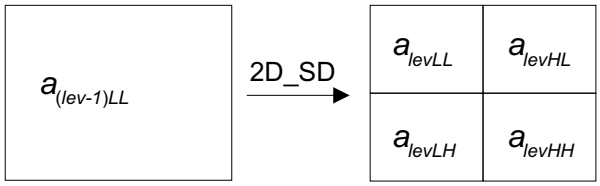

Figure 1.5: One-level decomposition — 2D_SD procedure

it performs *vertical* and *horizontal subband decompositions* of a two-dimensional array of coefficients. Then, the coefficients are deinterleaved into a set of four subbands.

The vertical subband decomposition procedure VER_SD (see alg. 1.15) takes as input the two-dimensional array $a_{(lev-1)LL}(u, v)$ and modifies it into vertically filtered version $a(u, v)$ of the input array, column by column.

The horizontal subband decomposition procedure HOR_SD (see alg. 1.16) takes as input the two-dimensional array $a(u, v)$ and modifies it into horizontally filtered version of the input array, row by row.

After vertical and horizontal subband decompositions, the result must be *deinterleaved* into four independent arrays. The idea of the deinterleaving is taking even rows and columns and forming the array of levLL subband from them. Then the procedure is repeated for even rows and odd columns for levHL, odd columns and even rows for levLH and odd columns and odd rows for levHH subband. The whole procedure is presented at alg. 1.17. Note, that variables $\mathbf{A_b}, \mathbf{B_b}, \mathbf{C_b}, \mathbf{D_b}, \mathbf{E_b}$ and $\mathbf{F_b}$ are taken from table 1.7.

The horizontal and vertical subband decompositions procedures call another procedure, 1D_SD. This procedure is composed of two parts: 1D_EXTD, which makes a *periodic symmetric extension* of the signal, and 1D_FILTD, which is basically a *lifting-based filtering* [5] and *scaling*.

Algorithm 1.13 Forward discrete wavelet transform

Input: $I(x, y),\ N_L$

Output: $a_b(u_b, v_b)$

$lev \leftarrow 1;\ a_{0LL}(u, v) \leftarrow I(u, v)$

while $lev \leq N_L$ **do**

$(a_{levLL}, a_{levHL}, a_{levLH}, a_{levHH}) \leftarrow$2D_SD$(a_{(lev-1)LL}, u_0, u_1, v_0, v_1)$

$lev++$

end while

Algorithm 1.14 2D_SD

Input: $a_{(lev-1)LL},\ u_0,\ u_1,\ v_0,\ v_1$

Output: $a_{levLL},\ a_{levHL},\ a_{levLH},\ a_{levHH}$

$a \leftarrow$VER_SD$(a_{(lev-1)LL}, u_0, u_1, v_0, v_1)$ {alg. 1.15}

$a \leftarrow$HOR_SD(a, u_0, u_1, v_0, v_1) {alg. 1.16}

$(a_{levLL}, a_{levHL}, a_{levLH}, a_{levHH}) \leftarrow$2D_DEINTERLEAVE$(a, u_0, u_1, v_0, v_1)$

Algorithm 1.15 VER_SD

Input: $a(u,v), u_0, u_1, v_0, v_1$

Output: $a(u,v)$

$u \leftarrow u_0;\ i_0 \leftarrow v_0;\ i_1 \leftarrow v_1$

repeat

$X(v) \leftarrow a(u,v)$

$Y(v) \leftarrow$1D_SD$(X(v), i_0, i_1)$

$a(u,v) \leftarrow Y(v)$

$u++$

until $u \geq u_1$

Algorithm 1.16 HOR_SD

Input: $a(u,v), u_0, u_1, v_0, v_1$

Output: $a(u,v)$

$u \leftarrow v_0;\ i_0 \leftarrow u_0;\ i_1 \leftarrow u_1$

repeat

$X(u) \leftarrow a(u,v)$

$Y(u) \leftarrow$1D_SD$(X(u), i_0, i_1)$

$a(u,v) \leftarrow Y(u)$

$v++$

until $v \geq v_1$

subband b	A	B	C	D	E	F
levLL	$\lceil u_0/2 \rceil$	$\lceil v_0/2 \rceil$	0	0	$\lceil u_1/2 \rceil$	$\lceil v_1/2 \rceil$
levHL	$\lfloor u_0/2 \rfloor$	$\lceil v_0/2 \rceil$	1	0	$\lfloor u_1/2 \rfloor$	$\lceil v_1/2 \rceil$
levHL	$\lceil u_0/2 \rceil$	$\lfloor v_0/2 \rfloor$	0	1	$\lceil u_1/2 \rceil$	$\lfloor v_1/2 \rfloor$
levHH	$\lfloor u_0/2 \rfloor$	$\lfloor v_0/2 \rfloor$	1	1	$\lfloor u_1/2 \rfloor$	$\lfloor v_1/2 \rfloor$

Table 1.7: Expressions for subbands used in 2D_DEINTERLEAVE

Algorithm 1.17 2D_DEINTERLEAVE

Input: $a(u,v), u_0, u_1, v_0, v_1$

Output: $a_{levLL}(u,v), a_{levHL}(u,v), a_{levLH}(u,v), a_{levHH}(u,v)$

for all *subbands* $b = [levLL, levHL, levLH, levLL]$ **do**
  $v_b \leftarrow \mathbf{B_b}$ {see table 1.7}
  repeat
    $u_b \leftarrow \mathbf{A_b}$
    repeat
      $a_b(u_b, v_b) = a(2u_b + \mathbf{C_b}, 2v_b + \mathbf{D_b})$
      u_b++
    until $u_b \geq \mathbf{E_b}$
    v_b++
  until $v_b \geq \mathbf{F_b}$
end for

The extension can be expressed by the equation

$$\begin{aligned} Y_{ext}(i) &= i_0 + \min(\mod(i - i_0, 2(i_1 - i_0 - 1)), 2(i_1 - i_0 - 1) - \\ &\quad - \mod(i - i_0, 2(i_1 - i_0 - 1))). \end{aligned} \tag{1.10}$$

subband i_0	i_{left}
even	4
odd	3

subband i_1	i_{right}
odd	4
even	3

Table 1.8: Extensions to the left and right

The *periodic symmetric extension* of the input X adds i_{left} coefficients to the left and i_{right} coefficients to the right side. This step is needed to enable filtering at both boundaries of the signal. The scheme of the periodic symmetric extension is depicted in fig. 1.6.

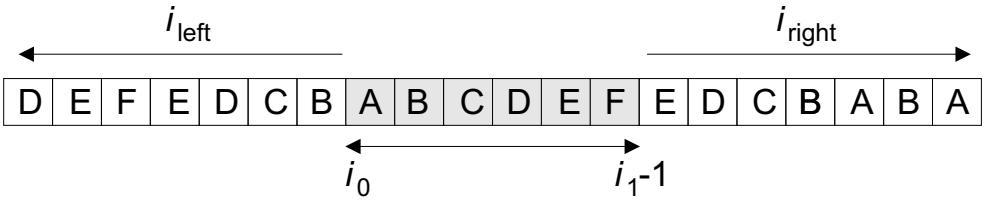

Figure 1.6: Periodic symmetric extension of signal

parameter	approx. value
α	-1.586134342059924
β	-0.052980118572961
γ	0.882911075530934
δ	0.443506852043971
K	1.230174104914001

Table 1.9: Lifting parameters for the 9-7 irreversible filter

Variables i_0 and i_1 are inputs of the extension procedure. i_0 means the first coefficient and $i_1 - 1$ the last coefficient of the input. Their values are fixed and they are enumerated for the case of the irreversible 9-7 encoder in table 1.8.

The output of the extension is filtered by the lifting-based filter. The filter is the core of the wavelet transform. Alg. 1.18 contains all operations which must be performed and table 1.9 is a list of approximate values of the parameters. The standard defines the parameters exactly. Variables i_0 and i_1 are again inputs of the filter. All $i_1 - i_0$ coefficients form the output Y of the filter and also the output of the SD_1 procedure.

1.9 DC level shifting and multiple component transform

DC level shifting and *multiple component transform* are two different procedures which are successively performed in the decoder and encoder. Because they are connected to each other and both are relatively simple, they are discussed together.

Fig. 1.7 shows where both procedures are used in a coding and decoding chain. However, the multiple component transform does not have to be used. For example a one-component image does not need it. Similarly, the DC level shifting does not have to be used either, if

Algorithm 1.18 Lifting based filter for encoder

Input: X_{ext}, i_0, i_1

Output: Y

for all $n : \left\lceil \frac{i_0}{2} \right\rceil - 2 \leq n < \left\lceil \frac{i_1}{2} \right\rceil + 1$ **do**

$Y(2n+1) \leftarrow X_{ext}(2n+1) + \alpha(X_{ext}(2n) + X_{ext}(2n+2))$

end for

for all $n : \left\lceil \frac{i_0}{2} \right\rceil - 1 \leq n < \left\lceil \frac{i_1}{2} \right\rceil + 1$ **do**

$Y(2n) \leftarrow X_{ext}(2n) + \beta(Y(2n-1) + Y(2n+1))$

end for

for all $n : \left\lceil \frac{i_0}{2} \right\rceil - 1 \leq n < \left\lceil \frac{i_1}{2} \right\rceil$ **do**

$Y(2n+1) \leftarrow Y(2n+1) + \gamma(Y(2n) + Y(2n+2))$

end for

for all $n : \left\lceil \frac{i_0}{2} \right\rceil \leq n < \left\lceil \frac{i_1}{2} \right\rceil$ **do**

$Y(2n) \leftarrow Y(2n) + \delta(Y(2n-1) + Y(2n+1))$

end for

for all $n : \left\lfloor \frac{i_0}{2} \right\rfloor \leq n < \left\lfloor \frac{i_1}{2} \right\rfloor$ **do**

$Y(2n+1) \leftarrow KY(2n+1)$

end for

for all $n : \left\lceil \frac{i_0}{2} \right\rceil \leq n < \left\lceil \frac{i_1}{2} \right\rceil$ **do**

$Y(2n) \leftarrow (1/K)Y(2n)$

end for

unsigned samples are encoded or decoded. Use of both procedures is signaled in appropriate markers and marker segments.

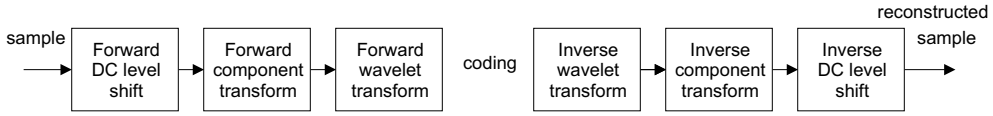

Figure 1.7: DC level shift and component transform in the coding process

If forward DC level shifting is used in the encoder, all samples $I(x, y)$ of the component are level shifted by subtracting the same quantity from each sample

$$I(x, y) = I(x, y) - 2^{shift}, \tag{1.11}$$

where $shift$ is written into the bitstream in an appropriate marker. It is obvious that this operation is always reversible and the decoder works exactly in the opposite way.

The multiple component transform is a bit more complicated issue. Two versions of this transform are used. The reversible one is performed when the reversible 5-3 wavelet transform is used. In this thesis, only the irreversible multiple component transform (ICT) will be described. It shall be used only with the 9-7 irreversible wavelet transform. The ICT decorrelates the first three components of an image and helps to achieve a better compression ratio. The three inputting components must have the same separation on the reference grid (see section 1.4) and the same bitdepth. The relationship between the components and the reference grid is also signaled in the bitstream in an appropriate marker.

If the first three components are red, green and blue components, the ICT is the $\mathrm{YC}_b\mathrm{C}_r$ transform.

The forward ICT is performed before forward wavelet transform and is computed for image component samples $I_0(x, y)$, $I_1(x, y)$, $I_2(x, y)$ as follows:

$$Y_0(x, y) = 0.299I_0(x, y) + 0.581I_1(x, y) + 0.114I_2(x, y) \tag{1.12}$$

$$Y_1(x, y) = -0.16875I_0(x, y) - 0.33126I_1(x, y) + 0.5I_2(x, y) \tag{1.13}$$

$$Y_2(x, y) = 0.5I_0(x, y) - 0.41869I_1(x, y) - 0,08132I_2(x, y) \tag{1.14}$$

The inverse ICT is performed after the inverse wavelet transform and is computed as

follows:

$$I_0(x,y) = Y_0(x,y) + 1.402Y_2(x,y) \quad (1.15)$$

$$I_1(x,y) = Y_0(x,y) - 0.34413Y_1(x,y) - 0.71414Y_2(x,y) \quad (1.16)$$

$$I_2(x,y) = Y_0(x,y) + 1.772Y_1(x,y) \quad (1.17)$$

1.10 Coding with regions of interest

The *region of interest* (ROI) technology will be described here only very shortly because it was not a part of the implemantation of the JPEG-2000 based codec (see section. 3.1).

As in other sections, the standard describes a norm only for the decoder. However, the possible implementation of the encoder will be discussed here. Similarly to previous sections, only irreversible compression case is interesting for this thesis.

The ROI is a part of an image that is coded earlier in the codestream than the rest of the image (the background). The information associated with the ROI preceedes the information associated with the background. The transform coefficients in the ROI are usually encoded with better quality.

The standard currently supports two methods to code the ROIs: the general scaling based method (GSBM) [6] and the maximum shift (Maxshift) method [3, annex H].

The *GSBM* moves the bitplanes of selected coefficients to higher bitplanes. A scaling value s determines how many bitplanes higher the ROI will be above the background. (see fig. 1.8). The main drawback of the GSBM is the need to encode the shape of the ROI.

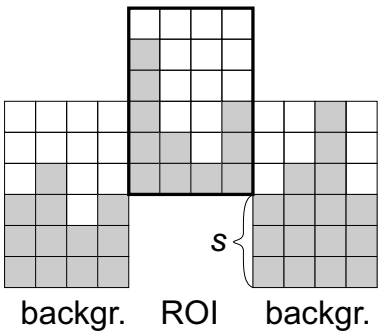

Figure 1.8: ROI, General scaling based method

The *Maxshift* method does not need to transmit the shape of the ROI. The quantized transform coefficients outside of the ROI are scaled down so that the bits associated with

the ROI are placed in higher bitplanes than the background (see fig. 1.9). It is necessary to save only the scale value s to the bitstream in an appropriate marker segment. The scale value must be

$$s \geq \max(M_b), \tag{1.18}$$

where M_b is given by eq. 1.5, so that the decoder can recognize which coefficients belong to the ROI. The main drawback of this method is that it is not possible to control the priority of the ROI. The background will be decoded after the ROI has been fully decoded. However, the quality of ROI does not have to be the same as the quality of the background because the accuracy of background coefficients can be sacrifized during encoding.

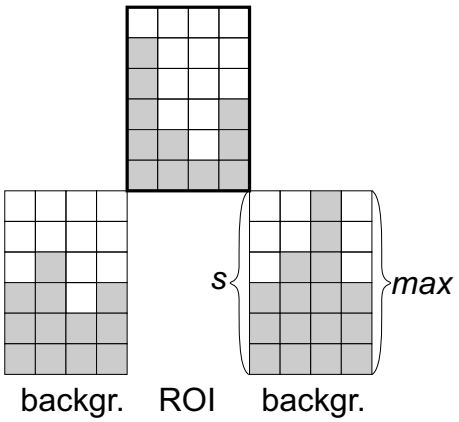

Figure 1.9: ROI, Maximum shift method

The encoder is entirely free to choose the coefficients which will be scaled. However, due to the nature of wavelet filters, the encoder should select them in such a way that inverse wavelet transform will always be able to use scaled coefficients when it needs to reconstruct the decomposition level from all of its subbands.

For the 9-7 irreversible filter, the rules can be easily obtained from the equations used in alg. 1.18. Dependencies of odd and even coefficients of the decomposition level on the coeffecients of the subbands are depicted in fig. 1.10.

1.11 JP2 file format syntax

Description of the JP2 file format, which provides necessary data structures for storing *metadata* in association with the JPEG-2000 codestream, will be given very briefly in this

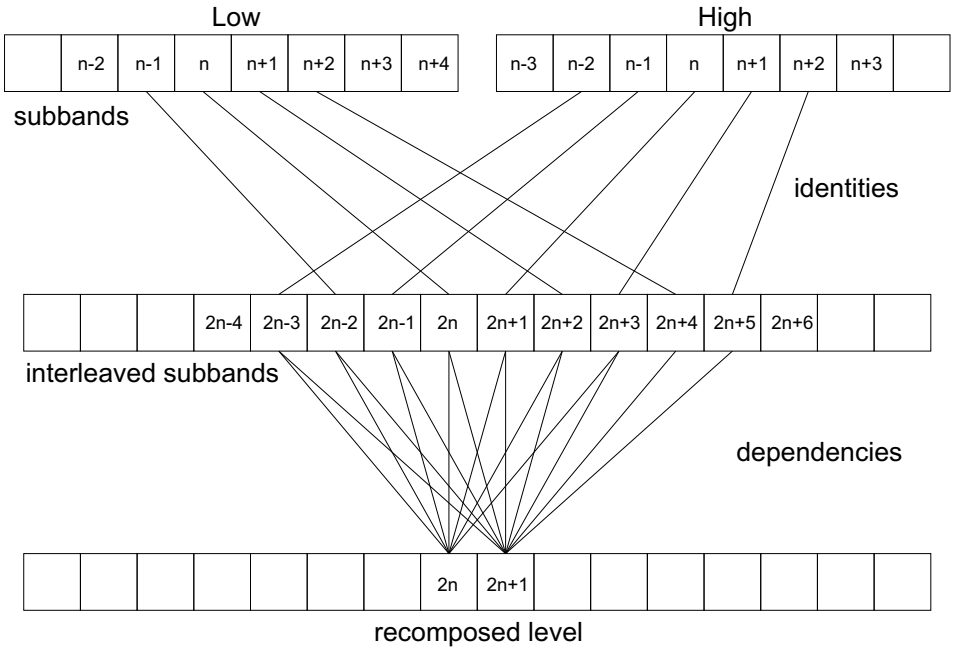

Figure 1.10: 9-7 irreversible filter dependencies

section because its implementation was not a goal of this thesis. The JP2 format is optional. Applications do not have to use it.

JP2 files are identified using several mechanisms. On traditional computer file systems, the names of JP2 files should include a file extension ".jp2". It might be interesting to note here that [7] registers the MIME type "image/jp2" for JP2 file format.

All information contained in the JP2 file is encapsulated in a collection of building blocks called *boxes*. Boxes define what information may be stored within them. Some boxes may contain other boxes and some of them are independent. Some boxes are mandatory, others are optional (marked as *Opt*). However, optional boxes can contain boxes which are mandatory, if and only if the optional box exists.

The order of boxes is not completely strict. Nevertheless, the first box in the file is the JPEG-2000 Signature box immediatelly followed by File Type box. JP2 header box shall appear before the Contiguous Codestream box. Other boxes, which are not defined in the standard, can appear between the boxes. All data in JP2 file must be in the box format.

The list of the boxes follows. More information about JP2 file format can be found in [3, annex I].

- **The JPEG-2000 Signature box**
- **File Type box**
- **JP2 Header box**
  - **Image Header box**
  - **Bits Per Component box**
  - **Colour Specification box 0**
    ⋮
  - Colour Specification box $n-1$ (*Opt*)
  - Palette box (*Opt*)
  - Component Mapping box (*Opt*)
  - Channel Definition box (*Opt*)
  - Resolution box (*Opt*)
    - Capture Resolution Box (*Opt*)
    - Default Display Resolution box (*Opt*)
- **Contiguous Codestream box 0**
  ⋮
- Contiguous Codestream box $m-1$ (*Opt*)
- IPR box (*Opt*)
- XML boxes (*Opt*)
- UUID boxes (*Opt*)
- UUID Info boxes (*Opt*)
  - UUID List box
  - Data Entry URL box

Chapter 2

Ebcot optimization

This chapter describes an image compression algorithm EBCOT (*embedded block coding with optimized truncation*). Like its predecessors (for example [9] and [10]), the EBCOT algorithm uses a wavelet transform to generate the subband samples which are to be quantized and coded. A dyadic decomposion structure attributed to Mallat [11] is typically used but other decompositions are also supported.

The algorithm produces highly resolution and SNR *scalable* bitstream. The bitstream is resolution scalable if it contains distinct subsets $\mathcal{B}_l$ representing each successive resolution level $\mathcal{L}_l$. The bitstream is SNR scalable if it contains distinct subsets $\mathcal{B}_q$, such that $\cup_{k=0}^{q}\mathcal{B}_k$ altogether represent the samples from all subbands at some quality (SNR) level q. If the bitstream holds both properties, then it is both resolution and SNR scalable. A key advantage of such a scalable compression is that the target bitrate or reconstruction resolution need not be known at the time of compression. A natural consequence is that the image need not be compressed multiple times in order to achieve a target bitrate.

EBCOT partitions subbands into relatively small *codeblocks* B_i of size, for example 32×32 or 64×64. A highly scalable bitstream is generated for each codeblock. Each bitstream can be independently truncated to any of a collection of different legths R_i^n. Of course, the reconstructed image from these truncations is distorted. This distortion can be modelled by D_i^n. The indices i are related to the number of the codeblock and indices n to the number of the truncation point. The total number of truncation points can be quite high, even hundreds or thousands and not all combinations enable good enough reconstructed images.

Methods, how to generate the set of the truncation points, how to choose among them and how the EBCOT handles data in general, will be explained in following sections.

2.1 Efficient one-pass control

The EBCOT algorithm truncates each of the independent codeblock bitstreams after all the subband samples have been compressed to achieve a target bitrate $R^{\max}$. This is called *post-compression rate-distortion* (PCRD) optimization. The advantage of PCRD is its simplicity. The image is compressed only once when PCRD algorithm collects all necessary data to be able to decide where the bitstreams should be truncated. Also memory usage is highly reduced. It is not necessary to keep the whole image or any comparable amount of data in memory at the same time. Because wavelet transform and codeblock coding can be implemented incrementally, the use of memory is constrained only to a width or height of the image [12].

The only image data which must be buffered for PCRD optimization is the embedded block bitstreams. These bitstreams are generaly much smaller than the original image. Moreover, also the PCRD optimization can be implemented incrementally so that only a part of the compressed block bitstreams need to be buffered. EBCOT provides the availibility of finely embedded bitstreams and the use of small blocks of subband samples.

2.2 Feature-Rich Bitstreams

The simplest way to implement resolution scalable bitstream with possible random access is by concatenating suitable truncated representations of each codeblock B_i. Of course, sufficient auxiliary information to identify the truncation points, n_i and the corresponding lengths $R_i^{n_i}$ must be included. The produced bitstream is obviously resolution scalable because data representing each codeblock and hence the subbands and resolution levels are strictly delimitated. It is also apparently "random accessible" — it is possible to identify the region of interest within each subband and hence the codeblocks which are required for its reconstruction.

However, this simple implementation is not SNR scalable, even when it is composed of SNR scalable block bitstreams. It is not possible to stop a coding process at some point when a given quality q is achieved and to keep the optimal rate-distortion ratio. Therefore the EBCOT algorithm introduces *quality layers* $\mathcal{Q}_q$ collecting incremental contributions from various codeblocks in such a way that the codeblock contributions represented by layers $\mathcal{Q}_i$, $i = 0, 1, \ldots, q$, form a rate-distortion optimal representation of the image for each q.

The PCRD algorithm can provide this feature. It must be noted, that the total number

of quality layers $\mathcal{Q}_q$ is limited. That is why the representation of the image is only approximately rate-distortion optimal. As the number of layers increases, the optimization gets better. However, the layers come also with some additional information to identify the size of each codeblock's contribution to the layer. This information must also be stored in the bitstream. Because it is in general substantially redundant information, EBCOT uses also a *second tier* coding to compress it (see fig. 2.1).

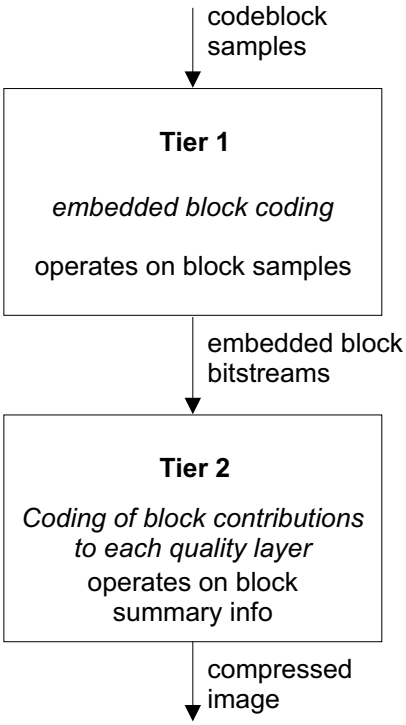

Figure 2.1: Two-tiered EBCOT coding

2.3 Rate-distortion optimization

The EBCOT partitions the subbands into collection of codeblocks B_i. Their embedded bitstreams may be truncated to rates R_i^n. The contribution from B_i to distortion of the reconstructed image is denoted D_i^n for each truncation point n. It is assumed that the relevant distortion metric is additive

$$D = \sum_i D_i^{n_i}, \tag{2.1}$$

where D denotes overall image distortion and n_i represents the truncation point selected for codeblock B_i. Different metrics can be used at this point. Only the simplest one will be mentioned in this thesis — *Mean Square Error* (MSE) metrics. The MSE approximation is obtained by

$$\hat{D}_i^n = w_{b_i}^2 \sum_{k \in B_i} \left(\hat{s}_i^n [k] - s_i [k]\right)^2 . \tag{2.2}$$

where $s_i[k]$ denotes the two-dimensional sequence of subband samples in codeblock B_i, $\hat{s}_i^n[k]$ denotes the quantized representation of these samples associated with truncation point n and w_{b_i} denotes the L2-norm of the wavelet basis functions for the subband b_i to which codeblock B_i belongs. This approximation is valid if the wavelet transform's basis functions are orthogonal or the quantization errors of the samples are uncorrelated.

Finding the optimal selection of the truncation points n_i means minimizing a distortion on the given bit-rate

$$R^{\max} \geq \sum_i R_i^{n_i}. \tag{2.3}$$

The optimization procedure comes from [13] and it will be described here. The set $\left\{n_i^\lambda\right\}$ which minimizes

$$D(\lambda) + \lambda R(\lambda) = \sum_i (D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda}) \tag{2.4}$$

for some *lambda* is optimal in the sense that the distortion cannot be reduced without also increasing the overall rate and vice-versa. If a value of λ, whose truncation points minimizing 2.4 yield $R(\lambda) = R^{\max}$, is found, then this set of truncation points has to be an optimal solution to our rate-distortion optimization problem. Because only a discrete set of truncation points is available, it is not possible to find a value of λ for which $R(\lambda)$ is exactly equal to $R^{\max}$. Anyway, since a lot of truncation points are used by EBCOT, it is sufficient in practice to find the smallest value of λ such that $R(\lambda) \leq R^{\max}$.

Looking for the optimal truncation points n_i^λ for any given λ is performed very efficiently. Only small amount of summary information collected during the generation of each codeblock's embedded bitstream is needed. In fact, the whole procedure consists of independent minimization problems of each codeblock B_i. The algorithm for finding the truncation point n_i^λ minimizing $D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda}$ is introduced in alg. 2.1.

Before executing this algorithm, the subset $\mathcal{N}_i$; $\mathcal{N}_i \subseteq \left\{n_i^\lambda\right\}$ of acceptable truncation points must be found in order to skip all truncation points out of a convex hull. The convex hull ensures that no point out of it produces better results than the points in it, that $\mathcal{N}_i$ is unique and that the largest set of truncation points which can be used by alg. 2.1.

Algorithm 2.1 Finding a truncation point in codeblock i

{Alg. produces an index n of a truncation point
which is optimal for a given λ}
init $n_i^\lambda = 0$ {i is a codeblock's number}
for $j = 1, 2, 3, \ldots$ **do**
{j is a truncation point's number}
$\Delta R_i^j \leftarrow R_i^j - R_i^{n_i^\lambda}$
$\Delta R_i^j \leftarrow D_i^{n_i^\lambda} - D_i^j$
if $\dfrac{\Delta D_i^j}{\Delta R_i^j} > \lambda$ **then**
$n_i^\lambda \leftarrow j$
end if
end for

Firstly, a pair of definitions should be given. An enumeration of the acceptable truncation points is defined as $j_1 < j_2 < \ldots$. The corresponding rate-distortion *slopes* to each neighbouring acceptable truncation points are given by

$$S_i^{j_k} = \frac{\Delta D_i^{j_k}}{\Delta R_i^{j_k}} \tag{2.5}$$

where $\Delta D_i^{j_k} = D_i^{j_{k-1}} - D_i^{j_k}$ and $\Delta R_i^{j_k} = R_i^{j_k} - R_i^{j_{k-1}}$. The slopes must be strictly decreasing in order that the points create the convex hull,

$$S_i^{j_{k+1}} < S_i^{j_k}. \tag{2.6}$$

For example, if $S_i^{j_{k+1}} \geq S_i^{j_k}$ then the truncation point j_k cannot be selected by alg. 2.1. When the set $\mathcal{N}_i$ is restricted, alg. 2.1 becomes a trivial selection of maximum

$$n_i^\lambda = \max(j_k \in \mathcal{N}_i;\ \mathcal{S}_i^{j_k} > \lambda). \tag{2.7}$$

Fig. 2.2 shows how the convex hull of the truncation points approximates the rate-distortion curve.

The EBCOT algorithm is usually implemented in such a way that $\mathcal{N}_i$ is determined immediately after the bitstream for B_i has been produced. The rates $R_i^{j_k}$ and slopes $S_i^{j_k}$ for each $j_k \in \mathcal{N}_i$ are stored until the last codeblock is compressed. At this time, the optimal λ and the set of optimal n_i^λ for this λ can be straightforwardly found. It is important to note that the distortion values need not be kept.

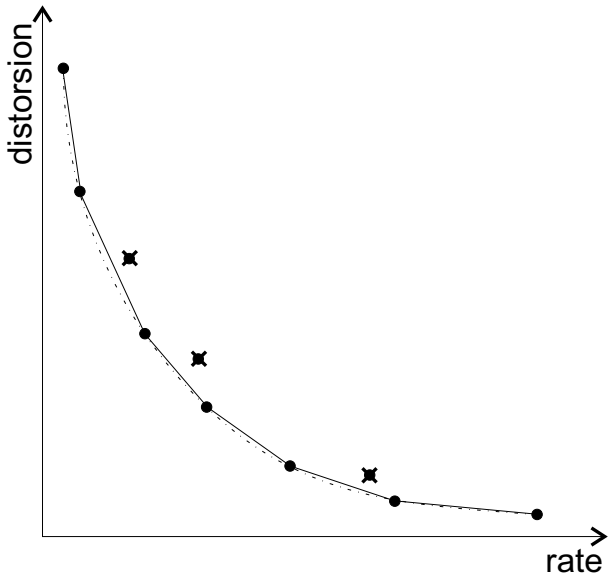

Figure 2.2: The rate-distortion curve approximated by the convex hull of the truncation points.

2.4 Additional notes to EBCOT algorithm

The paper about EBCOT algorithm [8] discusses also many other issues needed to output the final bitstream. They will not be widely discussed in this thesis because an essential part of the EBCOT algorithm is its codeblock optimization. However, at least basics should be mentioned here for completeness.

The codeblock coding is based on LZC algorithm [14]. The codeblock is further partitioned into *sub-blocks*. In order to gain more truncation points, the *fractional bitplanes* are used.

The codeblock is coded bitplane by bitplane using similar techniques like the JPEG-2000 (see p. 12). An idea of *deadzone quantizer* is behind the quantization in the EBCOT algorithm (see fig. 2.3). The deadzone quantizer has uniformly spaced thresholds, except the interval around zero, which is twice as large.

The entropy coding consists of four different primitives — primitive coding operations

- zero coding (ZC) (9)
- runlength coding (RLC) (1)
- sign coding (SC) (5)

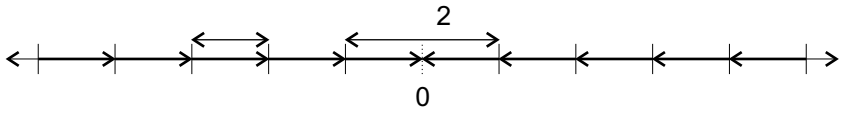

Figure 2.3: Deadzone quantizer

- sign coding (MR) (3)

The total number of contexts which EBCOT uses is 18. The numbers of contexts are stated at each primitive in the list.

Chapter 3

Implementation

The goal of the implementation was to evaluate the codeblock optimization methods in the JPEG-2000 based codec. The JPEG-2000 standard does not specify the optimization procedures because they are part of the encoder which is not standardized.

The first step of the implementation was to prepare a JPEG-2000 based framework including nonoptimized encoder and decoder and then implement the codeblock optimization into the encoder. The author implemented only those procedures from the JPEG-2000 standard that are necessary to evaluate the perfomances of the optimization algorithm. The perfomance evaluation can be done with the partially implemented JPEG-2000 standard. The entire implementation of both encoder and decoder is summarized in fig. 3.1.

The functions `decoder` and `optencoder` are implemented as Matlab functions. They are written in C programming language, compiled by Matlab-provided `mex` compiler script. Some non-core features of the functions are implemented by calling inner Matlab routines. Similarly, the forward and reverse wavelet transforms are functions implemented in Matlab and the C program calls them whenever they are needed.

Matlab files `decoder.m` and `optencoder.m` provide only descriptions of these two functions and their arguments which can be read in the Matlab environment by command `help`.

3.1 Limitations of the implementation

Because the implementation does not need to contain all features of the JPEG-2000 standard, it is highly reduced and limited. The most important changes are mentioned in the following paragraphs.

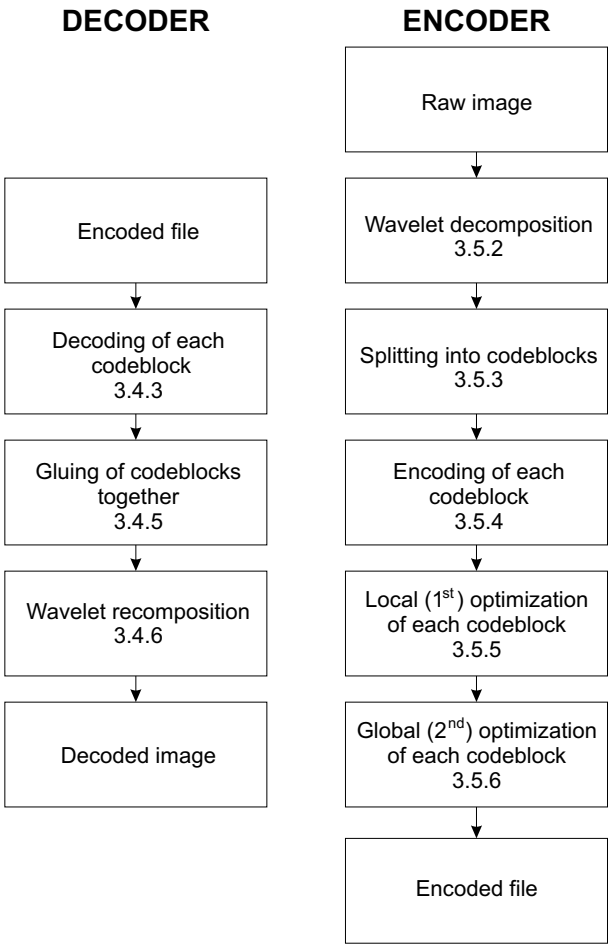

Figure 3.1: Flowcharts of implemented encoder and decoder with subsections of their description

The implementation does not support tiling of the image. It is equivalent to the situation when images are not tiled at all. Therefore the implementation can have problems with large images (over 1024 by 1024 pixels).

The height and width of the image can only be powers of two. Theoretically, the algoritm can handle any reasonable size of the image but this was not tested. The height and width must be greater than 2^n, where n is number of decomposition levels. The image need not be a square.

The only tested bitdepth was 8 bits per sample. The algorithm can theoretically process also any other different bitdepth. It would be limited only by the maximal possible transform

coefficient $2^{32} - 1$. However, the number is also limited by an extent of `uint8` Matlab data type.

The implementation can only work with one-component images. It means that only grey-scale images are permitted. For testing purposes, grey-scale images are sufficient. This component usually carries the most information.

The implementation does not map samples to the reference grid. Because only one-component images are used, the mapping can be virtually omitted. This corresponds to "one to one" mapping.

The implementation does not include two-tier coding and in consequence it does not provide SNR scalable bitstreams. This feature can be added by implementing quality layers (see sections 1.4 and 2.2). This feature is not necessary for testing purposes.

The error resilience features are not implemented because they are not necessary for testing purposes.

The codec does not support regions of interest. Again, this feature is not essential, although it would not be a difficult task to implement it.

The codec uses its own file format which is simple and sufficient to include all the features which are implemented.

Arithmetic coding is, of course, used in this implementation. Nevertheless, instead of the specific binary coder highly optimized for speed, which is introduced in section 1.5, the general arithmetic encoder is used (see [16] and subsection 3.7.1).

3.2 Interface

As it was mentioned above, the functions `decoder` and `optencoder` are called from Matlab environment. The names of the functions are derived from their roles. A prefix `opt` in the name of the encoder is an abbreviation of "optimized".

3.2.1 Decoder

The `decoder` function has the only parameter `arifilename` which is the name of the file which contains data in a format described in section 3.8. The parameter is a text string and it is mandatory.

The output argument `image` of the function `decoder` is a two-dimensional `uint8` matrix

of size (height, width) representing decoded image. It contains the samples of the image (pixels).

An example of usage:

```
result = decoder('jirka.ari');
```

This command reads and decodes the file `jirka.ari` and the result is stored in the Matlab matrix `result`.

3.2.2 Encoder

The `encoder` is called with several parameters affecting the encoding and optimization and two filenames used for input and output. All parameters are mandatory.

The `imagefilename` is a name of the image file which should be encoded or path to it. Any format, which can be read by Matlab, will be read also by the `encoder` function, for example jpeg, tiff, bmp, gif, pnm etc... The parameter is a text string.

The parameter `nl` of type `uint8` represents the number of levels in the wavelet decomposition which will be used within the encoding procedure. Constraints of this parameter are

$$1 \leq \texttt{nl} \leq 15 \tag{3.1}$$

$$\texttt{nl} \leq \log_2(width) - 2 \tag{3.2}$$

$$\texttt{nl} \leq \log_2(height) - 2 \tag{3.3}$$

where *width* and *height* are the dimensions of the image.

The parameter `cbp` of type `uint8` denotes the logarithm for a base 2 and the maximal height and width of codeblocks used for encoding optimization. The constraints of this parameter are

$$2 \leq \texttt{cbp} \leq 9 \tag{3.4}$$

$$\texttt{cbp} \leq \log_2(width) - 1 \tag{3.5}$$

$$\texttt{cbp} \leq \log_2(height) - 1 \tag{3.6}$$

where *width* and *height* are the dimensions of the image. The algorithm itself allows also nonsquare codeblocks but they were not tested because square codeblocks are sufficient enough for the purposes of algorithm testing.

The parameter `lambda` is the main parameter for optimization. It is a *slope* in the rate-distortion curve of each codeblock which should be found. A truncation point, belonging

to this slope, is then used as an optimal point. See section 2.3 for more information about `lambda`. The parameter `lambda` has no constraint. If `lambda` < 0 then the result image is not quantized.

The second and finer optimization (see subsection 3.5.6) is controlled by the parameter `secpar` of `uint8` type. This parameter is a number of neighbouring truncation points which will be used while searching for a better truncation point in the second optimization procedure. As greater `secpar` is given as better result can be found but the computing will take longer time. If `secpar` $= 0$, then no second optimization is performed, therefore the computation time is reduced to the minimum. Apparently, `secpar` ≥ 0.

The last parameter `arifilename` is the name of the output file or path to it. The format of the saved file is decodable by function `decoder` and is described in section 3.8.

The function `optencoder` has no output argument.

An example of usage:

```
optencoder('j.png', uint8(6), uint8(5), 4., uint8(2), 'j.ari');
```

This command compress the file `j.png` and stores the result into the file `j.ari`. The function uses 6 decomposition levels and codeblocks of size 32×32 coefficients. The main optimization parameter `lambda` is 4. The optimization will include also the second optimization procedure when two truncation points around the truncation point, which has been found in the first optimization procedure, are investigated.

3.3 Technical details of implementation

The implementation was written for Matlab of version 6.5.1.199709, Release 13 with Service Pack 1 and compiled by `mex` script using gcc compilator of version 3.3.1-2mdk for Mandrake Linux 9.2.

Most of the implementation is written in C programming language with standard C libraries except the code for the recomposition of the wavelet coefficient matrix. This part was implemented as a Matlab script and is derived from a work of Ioan Tăbuş. The arithmetic decoder, which is completely written in C language, is derived from an algorithm implemented by Gergely Korodi.

3.4 Image decoder

An idea of the decoder is quite simple, although it is taken from the quite complex JPEG-2000 standard. The overall idea of the procedure is presented in alg. 3.1. The algorithm is fast. The slowest part is the recomposition. The reason remains a bit unclear. Partially because it is really a complex procedure and partially because it is implemented as a Matlab script which is significantly slower than C code. An image of size 512×512 pixels is decoded by a computer with Pentium III class processor in less than a single second.

Algorithm 3.1 Implementation of the decoder

Input: name of the encoded file
Output: Matlab matrix of decoded image
  check input parameters
  read header
  for all codeblocks **do**
    read data of the current codeblock
    decode data of the current codeblock
  end for
  glue codeblocks altogether
  recompose image
  return recomposed matrix

3.4.1 Checking of input parameters

The only input parameter of the function `decoder` is checked in the beginning of the procedure. The check is simple, the parameter must be a Matlab vector of characters. The existence of the file is checked later while opening the file. If the file does not exist, the function is terminated with an error.

The content of the file is not checked or presumed. All information in the header of the file is checked during its processing. If any of the values in the header exceeds its limits, the function is terminated with an error as well. The reading of the header is more detailly described in the following subsection.

3.4.2 Header reading

All necessary information about the image except compressed data itself is read during the reading of the header. All the data is checked according to rules defined in section 3.8 in order to ensure decodability of the read file.

Within this phase, the decoder gets information about width and height of the image, number of decomposition levels used by the wavelet transform and maximal size of the codeblocks.

At this moment, the decoder already knows how the partition of the image looks. Therefore also the number of the codeblocks and their sizes are known.

The number of nonzero bitplanes and the position of the breakpoint in each codeblock is read and checked if they do not exclude each other. If yes, the decoder is terminated with an error.

After the header is read, the data structure can be prepared to read data for each codeblock from the arithmetic decoder.

3.4.3 Codeblock reading and decoding

The codeblocks are read one by one from the arithmetic decoder (detailed description is in subsection 3.7.2). The arithmetic decoder is reinitialized before each codeblock is going to be decoded.

If the file does not contain enough data, the algorithm is not able to recognize an unexpected end of the file. Since the algorithm reaches the end of the file, zeros are read and sent to arithmetic decoder all the time. The decoder supposes the unbroken input file. However, this uncorrectness was not a handicap during testing of the algorithm's performance.

A fundamental part of this procedure is a codeblock decoder. The codeblock decoder is implemented to realize the JPEG-2000 standard algorithms which are briefly introduced in section 1.6. It is composed of three passes: significance propagation, magnitude refinement and clean-up passes. The series of these three passes is executed for each bitplane. The highest bitplane is an exception — only the clean-up pass processes it.

The description of the decoder's passes and coefficient bit modelling would be practically only repeating the JPEG-2000 standard [3, annex D]. Therefore it is skipped in this thesis, although they are pivotal parts of the implementation. The reader can also revisit section 1.6 in order to get more information.

3.4.4 Dequantization

If the coefficients in the codeblock were quantized and thresholded after a certain bitplane, the decoding is stopped at the same point. The dequantization is performed after all data of codeblock is received and only if the coefficients were quantized. The reverse quantization can be expressed by the following formulas

$$|C_r| \;=\; \begin{cases} |C_q| & \text{if } q = 0 \\ |C_q| + 2^{q-1} & \text{if } q > 0 \end{cases} \tag{3.7}$$

where q is the quantization step, $q = 2^l$ where l is the least significant bitplane of the codeblock which is already completely processed.

Obviously, signs of the coefficients do not alternate during dequantization. The principles of the dequantizer are depicted in fig. 3.2. One can compare it with the encoder's deadzone quantizer in fig. 2.3.

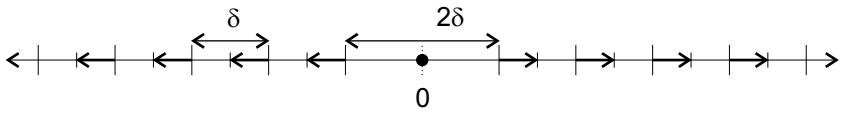

Figure 3.2: Dequantizer

3.4.5 Gluing of codeblocks together

When data of the codeblocks is read, processed by an arithmetic decoder and stored in memory, it is time to join them together in order that the wavelet decomposition can convert them from wavelet transform coefficients to image samples (pixels).

This algorithm of gluing is very simple. Actually, it involves only copying of memory blocks into new places. The main problem is to find the position and consequently also a size of each codeblock. This matter is discussed in the following paragraphs.

The future matrix of the wavelet transform coefficients is divided into subbands of decomposition levels. There are `nl` decomposition levels (`nl` is read from the header of the encoded file, see section 3.8). Each decomposition level *lev* includes four subbands (LL, HL, LH and HH). The LL subband of the the decomposition level *lev* is a complete decomposition level $lev - 1$. The exception is the LL subband of the decomposition level `nl`, which is not decomposed anymore.

During decoding, the decomposition levels are browsed from the decomposition level `nl` to the decomposition level 1. Firstly, the LL subband of the decomposition level `nl` is browsed. The rest is completely regular. The HL, LH and HH subbands (in this order) are browsed in each subband. The last browsed subband is the HH subband of the decomposition level 1.

The subbands of different decomposition levels have obviously different sizes. Because the `cbp` parameter — maximal size of codeblock (this parameter is read from the header of the encoded file, see section 3.8) — is not limited by the smallest subband, the subbands, which are smaller than the maximal size of the codeblock or have the same size, contain a single codeblock. Otherwise, the subband is divided into independent codeblocks. Clearly, the codeblocks cannot exceed the boundaries of the subbands. The codeblocks are ordered in each codeblock row by row.

When the decoder has found the position and size of each codeblock, the codeblocks are placed onto the correct places. The example of codeblocks' order is in fig. 3.3

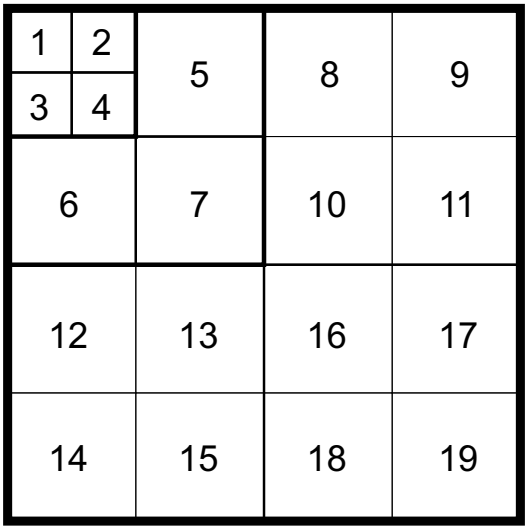

Figure 3.3: Order of codeblocks for `nl` $= 3$ and `cbp` $= \log_2 \frac{width}{4}$

3.4.6 Image recomposition

Up to now, the decoder has prepared the matrix of data which should be transformed now. The inverse 9-7 wavelet transform is used. The implemented encoder also uses 9-7 irreversible wavelet transform (see subsection 3.6.1). That is why the restored image samples will never be exactly the same as they were before the encoding, even when wavelet transform coefficients are not quantized. The codec is optimized for lossy, i.e. quantized, compression.

Additional information about the used inverse wavelet transform can be found in subsection 3.6.2.

3.5 Image encoder

The image encoder is based on the JPEG-2000 standard. It must generate the bitstream which is described in section 3.8. The procedure of encoding is summarized in alg. 3.2.

The image encoder is a more complex Matlab function than the `decoder`. The function is called `optencoder`. The encoder contains parts which have similar function like the decoder and, in addition to them, an encoder optimizing routine based on the Ebcot optimization (see chapter 2) which basically cuts a bitstream of each codeblock in order to achieve the highest PSNR for a given rate.

If one compares the Ebcot optimization and place where the first optimization is performed in this implementation, one must notice that a certain important property of the Ebcot optimization is not used. The codeblocks are not optimized immediately after the codeblock is read and encoded. The reason is that the second optimization can be executed only if all codeblocks are already processed by the first optimization. Running the first optimization during processing the codeblocks would be more efficient indeed but also a more complicated solution. Moreover, for the testing purposes this property is not crucial.

3.5.1 Checking of input parameters

The input parameters of the encoder are checked during initializing of encoder and loading the source image. Constraints of all parameters are mentioned in subsection 3.2.2.

In addition, the properties of the source image are checked. There must be only one component in the image, the bitdepth of the image must be 8 and the width and height of the image must be powers of 2.

Algorithm 3.2 Implementation of the encoder

Input: image file `imagefilename`, `nl`, `cbp`, `lambda`, `secpar`, `arifilename`
Output: file with a bitstream of encoded image
  check input parameters
  read original image from file `imagefilename`
  decompose image
  split coefficients into codeblocks
  for all codeblocks **do**
    encode data of the current codeblock
    store information about each truncation point of the bitstream
  end for
  for all codeblocks **do**
    choose the truncation point for the current codeblock {1^{st} optim.}
  end for
  if the second optimization is used **then**
    for all codeblocks **do**
      find better truncation point for the current codeblock {2^{nd} optim.}
    end for
  end if
  write header into file `arifilename`
  encode bitstream of all optimally truncated codeblocks
  save the created bitstream into file `arifilename`

3.5.2 Image decomposition

The first procedure of the encoder is a wavelet decomposition. The forward 9-7 wavelet transform is used. The used transform is irreversible. That is why the saved image will never be exactly the same as the source image, even when wavelet transform coefficients are not quantized. The codec is optimized for lossy, i.e. quantized, compression, thus advantages of lossy wavelet transform can be taken into account with no harm.

Additional information about the used forward wavelet transform can be found in subsection 3.6.1.

The result of the forward 9-7 wavelet transform is a Matlab matrix of real numbers. However, the codeblock encoding procedure processes only integer numbers. Thus the coefficients of the wavelet transform are converted before they are further processed.

3.5.3 Splitting into codeblocks

Splitting the matrix of wavelet coefficients into codeblocks is an opposite procedure to gluing of codeblocks together (see subsection 3.4.5). Similarly to gluing, splitting procedure browses the subbands of decomposition levels and codeblocks inside them. Because the order of the codeblocks must stay the same in the encoder like in the decoder, the method of browsing is identical.

The browser calls codeblock encoding procedure for each codeblock which is visited.

3.5.4 Codeblock encoding

Codeblock encoding is an essential part of the codec, its heart. This part is a practically complete implementation of section 1.6 except its subsection 1.6.5, which describes nonsubstantial peculiarities of the JPEG-2000 standard. Error resilience, vertically causal context and selective arithmetic coding bypass are not implemented. Reinitializing of arithmetic encoder is implemented in a different way and is mentioned in subsection 3.5.8. The codeblock encoding can also be considered as an opposite algorithm to codeblock decoding (see subsection 3.4.3).

Thus the codeblock encodes bitplane by bitplane in three passes: *significance propagation* (see subsection 1.6.1), *magnitude refinement* (see subsection 1.6.3) and *cleanup* pass (see subsection 1.6.4).

All bits which are encoded during this pass are sent with their contexts to the arithmetic encoder (see subsection 3.7.1) and also to bitstream in memory where they are stored in order to be reencoded again after the truncation points have been chosen. Reencoding is necessary due to properties of the chosen arithmetic encoder.

Before the first pass and then after each one, the information about truncation points is memorized. This information includes length of the encoded stream, which was generated from the beginning of the current codeblock encoding, *rate* - a ratio of *length* and codeblock's *area* (it is not necessary but handy), *mse* (also not necessary) and *slope* (see definition on the page 36). This information is later used for codeblock optimization.

3.5.5 First optimization of codeblocks

The algorithm introduced in subsection 3.5.4 can encode the image but it cannot decide how to choose data in order to generate more or less distorted image of demanded quality and/or

rate. In our context, choosing data means choosing truncation points for all codeblocks. This is provided by the optimization procedure. Actually, the truncation points can be chosen very naively. One possibility is simply not to use data for certain codeblocks, for example those from the end of the stream (the stream ends with the most down right codeblock in the HH subband of the first decomposition level). Another possibility is to truncate all codeblocks at the same point, for example after a certain number of truncation points from the beginning or before the end of the codeblock encoding. Apparently, these methods are not very effective.

The method, which was used in this implementation as the first step of optimization is based on the Ebcot optimization (see chapter 2). Every codeblock is processed independently. The collection of the truncation points is ordered by the rate. The convex hull of the points is found (see fig. 2.2). The slopes between each pair of successive truncation points on the convex hull are computed. The algorithm chooses the first point of the pair with the slope less than or equal to the parameter `lambda` as the optimal point (see alg. 2.1).

Note that the *rate* R_i is a length of the encoded bitstream of the current codeblock truncated at the current truncation point divided by the area of the codeblock. Similarly, the *distortion* D_i is a mean square error between the original codeblock and truncated codeblock.

The Ebcot optimization cuts the bitstreams of the codeblocks in different places. Therefore, the position of the truncation points must be specified in the encoded file. This is described in section 3.8.

3.5.6 Second optimization of codeblocks

The optimization of independent codeblocks is fast and gives good results. However, it works perfectly only if the wavelet transformation is orthonormal, which is not the case of used Daubechies 9-7 wavelet filter [15], even when this filter is very close to orthonormal. The consequence of the non-orthonormality is a different distortion of the recomposed image for the same error injected to different codeblocks.

In the implementation described in this thesis, this causes that the found truncation points are not optimal from a point of view of the recomposed image. Surely, it is possible to find the rate-distortion pair for every truncation point of every codeblock. Unfortunately, this means that the recomposition of the entire image is necessary for every truncation point. The wavelet recomposition is arithmetically too complex. Therefore, the performance of such an algorithm is very time consuming.

However, it is possible to assume that the true optimal truncation point is close to the point which was found by the first optimization. One can search for the optimal point only among the truncation point in a given small neighbourhood. This significantly reduces computation time. A radius of the neighbourhood is given by the parameter `secpar` (see subsection 3.2.2).

Note that in this case, the *rate* R_i is the length of the encoded bitstream of the entire image with the current codeblock truncated at the current truncation point divided by the area of the image. Similarly, the *distortion* D_i is a mean square error between the original image and image recomposed with the current truncated codeblock. If truncation points from both optimizations are compared, values of the rate and distortion do not have to be normalized because they are relative to areas to which they belong.

The quality of the investigated truncation point $[R_i, D_i]$ is measured in a rate-distortion plane by a projection p of the vector from the truncation point $[R_0, D_0]$, which was found by the first optimization, to the investigated truncation point on a perpendicular direction to the direction of `lambda` (see fig. 3.4):

$$p = \sin(\arctan(\texttt{lambda}))(R_0 - R_i) + \cos(\arctan(\texttt{lambda}))(D_0 - D_i) \tag{3.8}$$

The algorithm of the second optimization is summarized in alg. 3.3. One can notice that the optimization of the codeblock depends on results of the optimization of codeblocks which have been processed before the current one. This improves the accuracy of the result.

The algorithm significantly reduces computation time in comparison with a brute force algorithm proposed at the beginning of this subsection. However, it is still a very time consuming procedure. The number of the performed recompositions depends linearly on the number of codeblocks (which depends mainly on its size `cbp` as $2^{-2\texttt{cbp}}$ and also slightly on the number of decompostion levels `nl`) and the main parameter of the second optimization `secpar`. Because time needed for the browsing throught all subbands and codeblocks grows linearly with the number of codeblocks as well, the total time needed to perform the second optimization falls as $2^{-4\texttt{cbp}}$. A small parameter `cbp` significantly increases the computation time. Just for completeness, the number of codeblocks obviously depends linearly on both height and width of the image.

3.5.7 Quantization

Because dequantization is a part of the decoder, it is worth to mention the quantizer here. Actually, the forward quantization has happened when the bitstream was truncated which

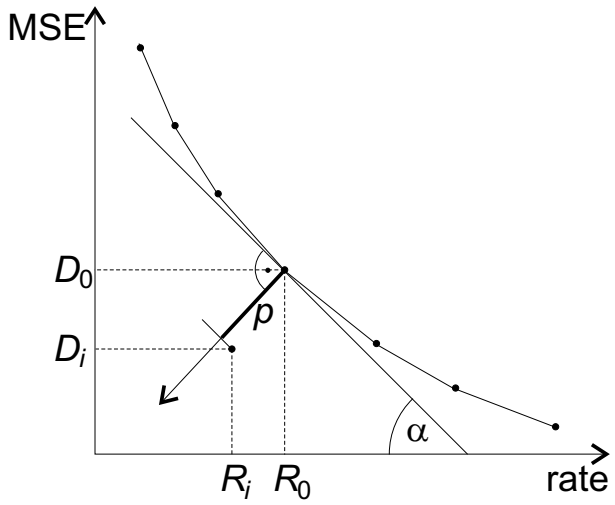

Figure 3.4: Quality of investigated truncation point corresponding to a slope `lambda` ($\tan(\alpha) =$ `lambda`) as a projection in R-D plane

corresponds to so called *deadzone quantizer* (see fig. 2.3). This quantizer can be expressed by the following formula

$$|C_q| = \left\lfloor \frac{C}{q} \right\rfloor, \tag{3.9}$$

where q is a quantization step and $q = 2^l$ where l is the least significant bitplane of the codeblock which is already completely processed. The chosen truncation point is immediately after any of the passes of this bitplane, but bitplane is completely processed only after finishing its cleanup pass. This gives better results than if l was the least significant bitplane of the codeblock at least partially processed.

Evidently, signs of the coefficients do not alternate during quantization.

3.5.8 Output to the file

The file contains two parts — a header and encoded data — as they are defined in section 3.8. The output procedure must satisfy it. In this section, the final arithmetic encoding is described.

All the codeblocks are reencoded again before the data is written to the output file because the used arithmetic encoder (see subsection 3.7.1) cannot continue in the stream which is already dumped. However, as it was empirically measured, the statistical model is better

Algorithm 3.3 The second optimization

Input: set of all codeblocks $\mathcal{C}_j$, set of all $\mathcal{T}_i^j$,
set of supposed optimal $\mathcal{S}_j; \mathcal{S}_j \in \mathcal{T}_i^j$, `secpar`
Output: set of nearly optimal $\mathcal{O}_j; \mathcal{O}_j \in \mathcal{T}_i^j$
for all codeblocks $\mathcal{C}_j$ **do**
for all $i;\ i \geq \mathcal{S}_j - \texttt{secpar}, i \leq \mathcal{S}_j + \texttt{secpar}$ **do**
$R_i \leftarrow$ *length* {*length* is the estimated length of the encoded file for image with $\mathcal{O}_j = i$}
recompose image
$D_i \leftarrow$ *mse* {*mse* is the error between the original image and recomposed image with $\mathcal{O}_j = i$}
end for
$\mathcal{O}_i^j \leftarrow i; \max_i(p)$ {see eq. 3.8}
end for

when the statistics are reinitialized after each codeblock is processed. The reinitialization is called from the output function during reencoding.

The whole bitstream is at first stored in the memory and then written into the output file in this implementation. It is not the most efficient solution but it is simple and satisfactory.

3.6 Wavelet transforms

3.6.1 Forward wavelet transform

Section 1.8 describes everything important of the forward wavelet transform as it was implemented. The irreversible 9-7 wavelet transform was chosen to be implemented because the optimization algorithm (see subsections 3.5.5 and 3.5.6) works with quantized coefficients anyway. Obviously, the irreversible transform is a more effective choice. The code itself is based on the Matlab code by Ioan Tăbuş. It was slightly changed because the original version did not support non-square images. The Matlab routines are called from C code using a standard API for Matlab executable (MEX) files.

3.6.2 Inverse wavelet transform

The inverse wavelet transform is completely described in [3, annex F]. The code is based on the code by Ioan Tăbuş and from technical point of view, everything, what was written in subsection 3.6.1 can be applied for it as well.

3.7 Arithmetic encoder and decoder

The arithmetic encoder and decoder are not coming from the JPEG-2000 standard at all. It was decided that an older implementation of the arithmetic encoder and decoder which was introduced in [16] is satisfactory. The code of both parts is based on the implementation of Gergely Korodi. His original implementation was only a kernel of the arithmetic encoder and decoder without a usable interface. It was used for several different purposes in the past (for example in [17]). However, the code had to be changed for each implementation. The implementation done for this thesis redesigned the original one in order to be used as a module of a more complex software package and it contains a needed application programming interface. Therefore it can be easily used for any application now.

A programming interface and the implementation of the arithmetic encoder and decoder are located in files `aricoder.h` and `aricoder.c`.

3.7.1 Arithmetic encoder

Principles of used arithmetic encoder and decoder were introduced in [16]. The implementation of the arithmetic encoder contains both a statistics collector and the arithmetic encoder itself. The statistics collector keeps frequencies of all symbols for each context in a stream. The arithmetic encoder then uses the collected statistics to produce the encoded bitstream.

The statistics collector and arithmetic encoder are initiated independently which must be handled by the calling function. The arithmetic encoder does not output every single bit immediately. It works with a register and outputs data only when the register is full. That implies that the register must be flushed out before the bitstream can be closed. The bitstream can also be dumped. This operation copies the already generated bitstream, flushes the register to the copy and restores the original register again.

It is important to mention the characteristics of the encoded bitstream here. The bitstream generated by this encoder suffers from an unpleasant property — when the bitstream

is dumped, it is not possible to continue encoding again. This causes a need to reencode the whole bitstream before it is stored in the output file.

Another unpleasant property is a generality of the encoder. The encoder was originally designed for the alphabet with arbitrary number of the symbols. For the case of binary alphabet used in this thesis, a specialized arithmetic encoder can be used. The encoder is general also from the point of view of the input data. The arithmetic encoder used in the JPEG-2000 standard respects the special property of the image data. The peculiarities of the image data are mostly included in the statistical model (see section 1.6) but the arithmetic encoder of the JPEG-2000 standard goes even further.

3.7.2 Arithmetic decoder

The idea of the arithmetic decoder is opposite to that of the encoder. The statistics collector works exactly in the same way. The statistics available to the encoder and decoder before encoding and decoding must be the same.

The significant difference is in the data structure. In contrast to the encoder, which encodes data to a bitstream in memory, the decoder reads data directly from a file.

The arithmetic decoder should be defined very correctly here because it determines how the bitstream must look to be decodable. However, optimized implementation of the arithmetic decoder was not a goal of this thesis. The thesis cannot include an exact description of the arithmetic decoder because of its limited format. One can say that the arithmetic decoder is defined by this reference implementation.

3.8 File format

The file format is defined from the side of the decoder. It consists of two parts — a header and data of truncated codeblocks. The fileformat of this implementation is completly different from the JPEG-2000 file format.

3.8.1 Header

The structure of the header is documented in table 3.1.

The first two items define the size of the image. Both dimensions must be powers of 2. Other dimensions were not tested because most of the test images have these dimensions.

Item	length in bits
Image *width*	16
Image *height*	16
`nl` -1	4
$\log_2($`xcbp` $-2)$	3
$\log_2($`ycbp` $-2)$	3
bits for passes p	3
numbers of passes	$p \cdot c$
bits for bitplanes b	3
numbers of bitplanes	$b \cdot c$

Table 3.1: Header of the file format

The next item is a number of the decomposition levels `nl` substracted by 1. The constraints for `nl` are

$$\texttt{nl} \leq \log_2(width) - 2, \tag{3.10}$$

$$\texttt{nl} \leq \log_2(height) - 2, \tag{3.11}$$

$$1 \leq \texttt{nl} \leq 15. \tag{3.12}$$

Parameters `xcbp` and `ycbp` limit the maximal size of codeblocks. Only the case `xcbp` = `ycbp` = `cbp` was tested in this implementation. Both items must satisfy

$$\texttt{cbp} \leq \log_2(width) - 1, \tag{3.13}$$

$$\texttt{cbp} \leq \log_2(height) - 1, \tag{3.14}$$

$$2 \leq \texttt{cbp} \leq 9. \tag{3.15}$$

At this point, the decoder computes a number of codeblocks c which partition the whole image (respectively its wavelet coefficients). The simplest way to do it is to browse throught all codeblocks of all subbands. The decoder may prepare the data structure for decoding at this point.

The next three bits p form the number of bits needed to express a number of passes which are encoded in the bitstream for each codeblock. A following block of $p \cdot c$ bits contains these numbers.

Similarly, next three bits b compose the number of bits needed to express an item from a following collection which contains numbers of bitplanes in each codeblock. This collection

is $b \cdot c$ bits long. This information is coded with the use of information from the previous block. The number of bitplanes B_c for a certain codeblock can be expressed as

$$B_c = R_c + \left\lceil \frac{P_c + 2}{3} \right\rceil, \tag{3.16}$$

where R_c is the read number (of length b bits) and P_c is the number of passes which are encoded in the bitstream for this codeblock. B_c is coded in order to avoid redundancy in the bitstream.

3.8.2 Data

Data in the second part of the file is ordered codeblock by codeblock. The order of codeblocks is described in subsection 3.4.5. See mainly fig. 3.3.

The end of the file is not recognized by the decoder. This complication comes from the nature of the used arithmetic decoder. If the decoder reaches the end of the file, all requested bits cannot be read. Instead, zeros fill the arithmetic decoder's register. There is no maximal number which can be requested by the decoder beyond the end of the file. Usually, only a couple of bytes are requested in reality.

Chapter 4

Experimental results

The optimized encoder described in chapter 3 was intensively tested and the results of these tests are presented in a shortened form here.

All plots, tables and other information in this chapter use the same convention of expressing the rate and quality of each image. The quality of the reconstructed image is always expressed by $PSNR$ (peak signal-to-reconstructed image measure)

$$PSNR = 20 \log_{10}(\frac{255}{\sqrt{MSE}}) \text{ [dB]} \tag{4.1}$$

where MSE (mean square error) is

$$MSE = \frac{\sum [u(x,y) - v(x,y)]^2}{N^2}, \tag{4.2}$$

where $u(x,y)$ and $v(x,y)$ are the original and reconstructed images and N is a total number of pixels of the image. The rate of the image is always expressed in bpp (bits per pixel). For example, the plots have the rate in bpp on the horizontal axis and quality in dB of $PSNR$ on the vertical one.

Abbreviation OPT(x) means that the implemented optimization algorithm was used with a parameter `secpar` $= x$. One must remember that `secpar` $= 0$ means that the second optimization was not used at all (see subsection 3.2.2).

4.1 Comparison of optimized and nonoptimized compression

Two versions of unoptimized image compression were considered. The first, UN1, cuts the stream of each codeblock after a certain and constant number of truncation points. The second, UN2, cuts the stream after a certain number of codeblocks, regardless of its position in the image. The codeblocks themselves are encoded till the last truncation points, that means almost losslessly.

All plots and images in this section are related to compressed images with parameters `nl`= 6, `cbp`= 6. The examples in the figures are chosen from the results of compression tests of various images. Also the secondary optimization is used in some cases. This does not affect the overall idea of the comparison because the results of compression with and without the second optimization are much closer to each other than to any of the unoptimized compressions.

Figures 4.1 and 4.2 show the importance of optimizing on two examples of rate-distortion plots of optimized and nonoptimized compression. The nonoptimized compression produces images of worse quality for the same rates. The rates for different methods were not unified for the tests. However, the results can be linearly interpolated without harming the final results. This was computed for the image Peppers and is presented in table 4.1.

rate	UN1	UN 2	OPT(0)	OPT(2)
0.1	15.09	23.74	29.38	29.57
0.2	17.63	25.77	32.29	32.40
0.3	18.80	27.20	33.74	33.84
0.4	19.98	28.38	34.81	34.90
0.5	21.47	29.18	35.36	35.50
0.6	22.96	29.74	35.83	36.02
0.7	23.97	30.19	36.44	36.62
0.8	24.26	30.64	37.05	37.22
0.9	24.56	31.11	37.66	37.82

Table 4.1: Image Peppers: interpolated PSNRs for different methods and various rates

Two images Lena in fig. 4.3 were created by nonoptimized (UN2) and optimized (OPT(0)) encoder. One can clearly recognize the difference in the quality at about the same rate.

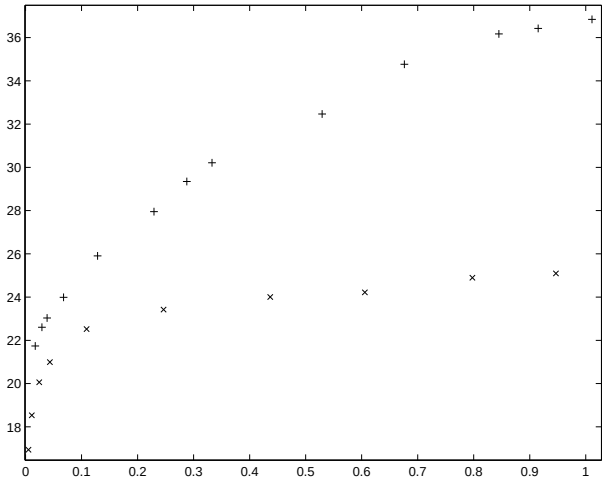

Figure 4.1: R-D plot of Barbara image: × UN2, + OPT(2)

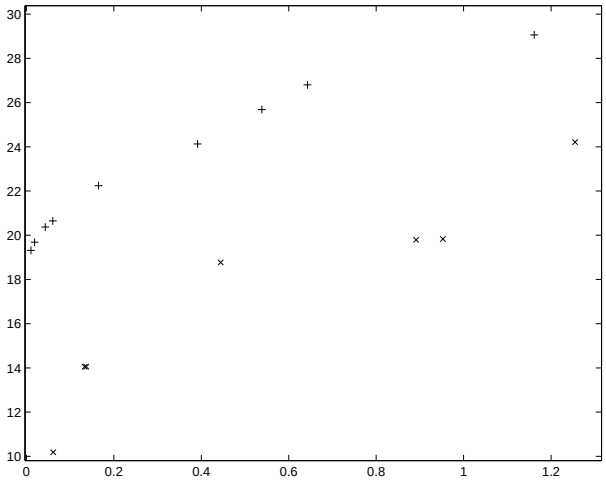

Figure 4.2: R-D plot of Mandrill image: × UN1, + OPT(0)

4.2 Comparison of optimized compression and SPIHT

The SPIHT encoder is often considered as a reference standard for wavelet based encoders. Therefore comparison with it must be dealt in this thesis. The rate-distortion plots were chosen to depict the similarities and distinctions between the behaviour of SPIHT and the implemented encoder.

All plots in this section (fig. 4.4–4.10) contain three groups of measurements. The SPIHT

Figure 4.3: Lena images: left UN2: rate 0.042, PSNR 22.55; right OPT(0) at `lambda` = 2000: rate 0.043, PSNR 26.63

reference is represented by lines and the implemented codec by crosses — "×" for encoding without the second optimization and "+" for encoding with the second optimization enabled and `secpar` = 2.

Common parameters `nl` = 6 and `cbp` = 6 were chosen for the measurements of the implemented codec in this section with the exception of Circles image when `nl` = 4 was used because this image is smaller than the others.

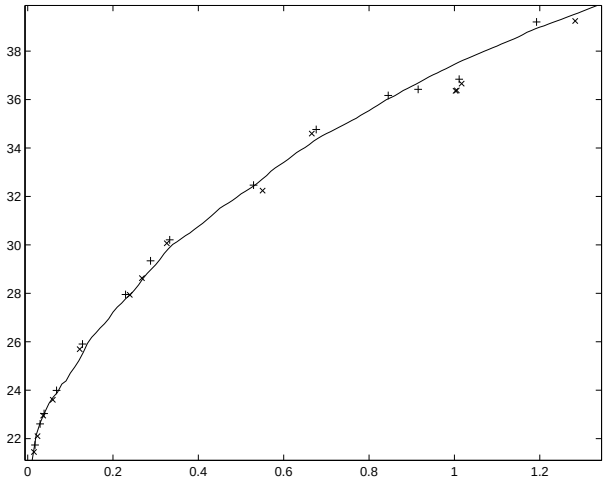

Figure 4.4: R-D plot of Barbara image: × OPT(0), + OPT(2), — SPIHT

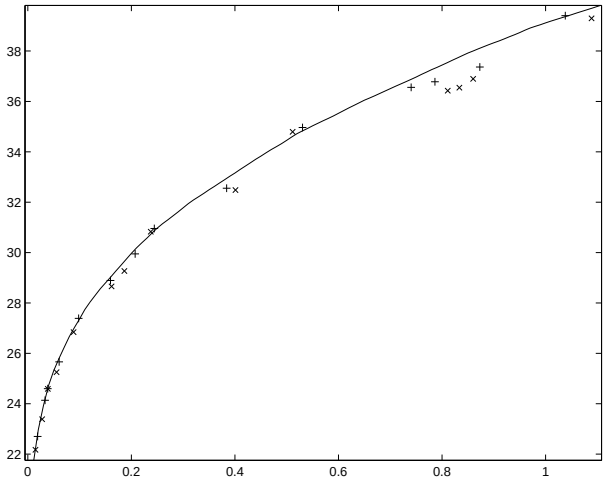

Figure 4.5: R-D plot of Boat image: × OPT(0), + OPT(2), — SPIHT

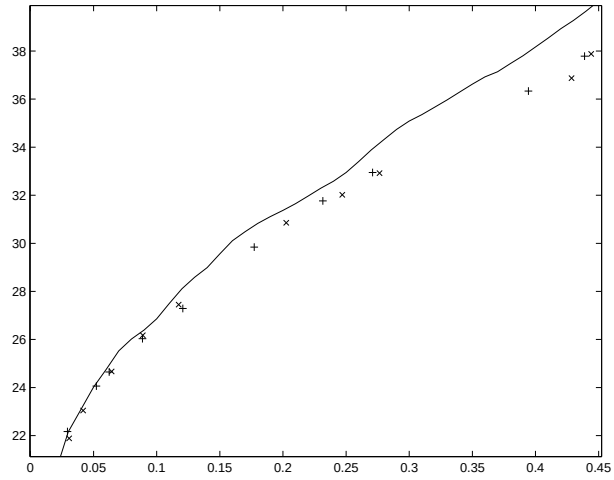

Figure 4.6: R-D plot of Circles image: × OPT(0), + OPT(2), — SPIHT

It is clearly visible that the implemented optimized encoder produces the streams of about the same quality as SPIHT. In some cases, especially for lower rates, the implemented encoder is better. On the other hand, one can recognize encoder's problems at higher rates. The encoder is significantly worse than SPIHT in the test of Circles image.

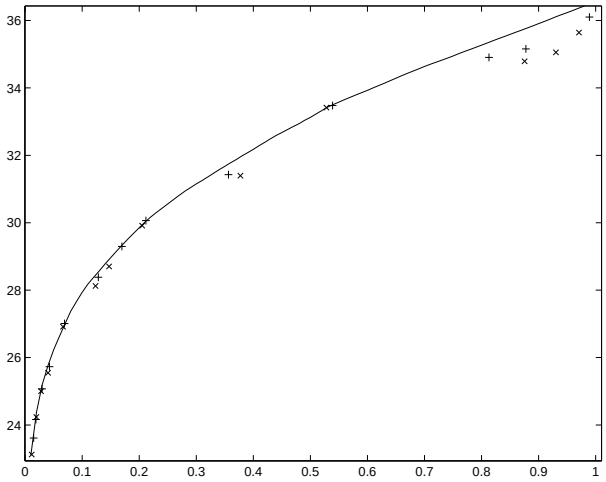

Figure 4.7: R-D plot of Goldhill image: × OPT(0), + OPT(2), — SPIHT

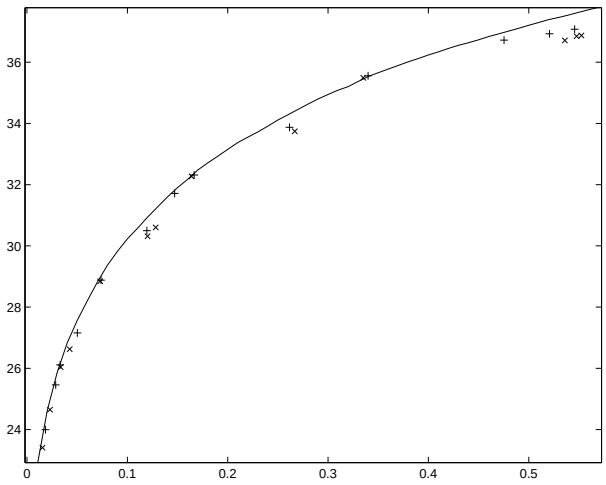

Figure 4.8: R-D plot of Lena image: × OPT(0), + OPT(2), — SPIHT

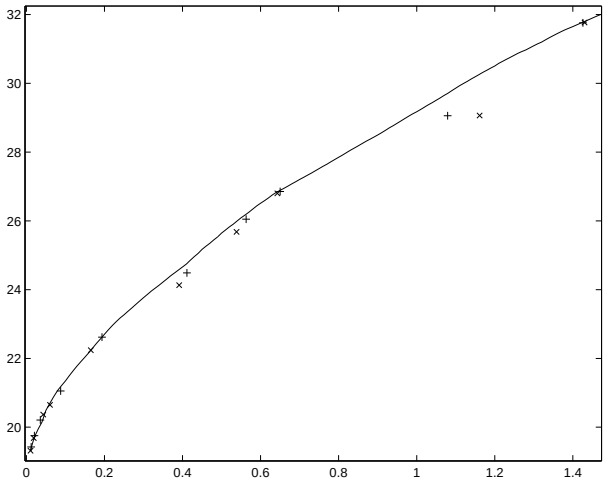

Figure 4.9: R-D plot of Mandrill image: × OPT(0), + OPT(2), — SPIHT

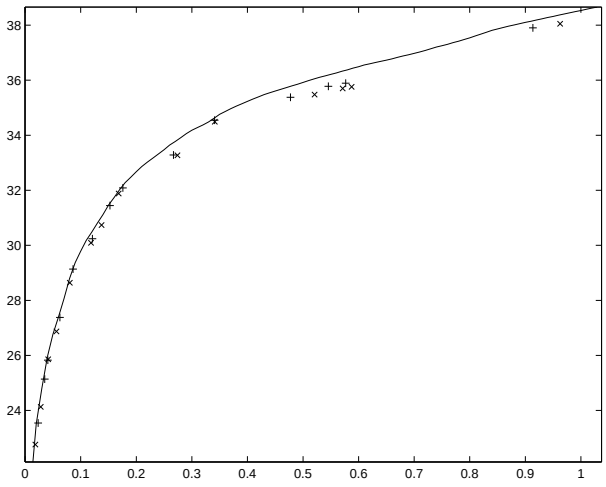

Figure 4.10: R-D plot of Peppers image: × OPT(0), + OPT(2), — SPIHT

4.3 Comparison of optimized codec and Jasper

An essential part of the tests must be a comparison with another JPEG-2000 codec. Jasper serves as a reference implementation of the JPEG-2000 standard, Part-1 standard. The comparison is, in fact, quite complicated because Jasper has more functions than only to compress the image and also has a different way to compute the statistical models (see section 1.5). In addition, the second tier optimization is implemented in Jasper, while it is not in the simplified version of the JPEG-2000 standard which was implemented here.

All plots in this section (fig. 4.11 – 4.17) contain three groups of measurements. The Jasper reference is represented by lines and the implemented codec by crosses — "×" for encoding without the second optimization and "+" for encoding with the second optimization enabled and `secpar` = 2.

Common parameters `nl` = 6 and `cbp` = 6 were chosen for the measurements of the implemented codec in this section with the exception of Circles image when `nl` = 5 and `cbp` = 5 were used because this image is smaller than the others.

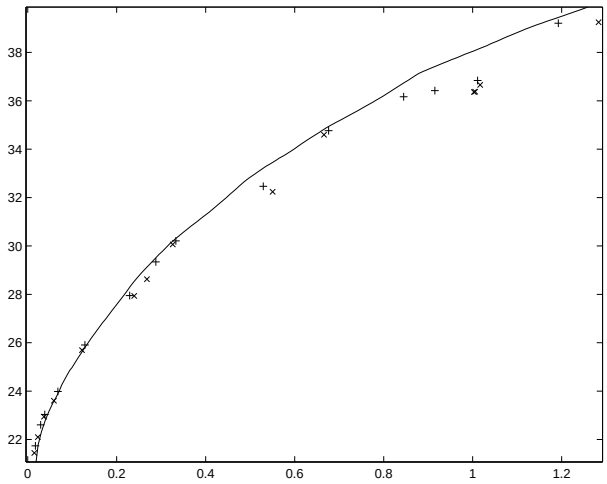

Figure 4.11: R-D plot of Barbara image: × OPT(0), + OPT(2), — Jasper

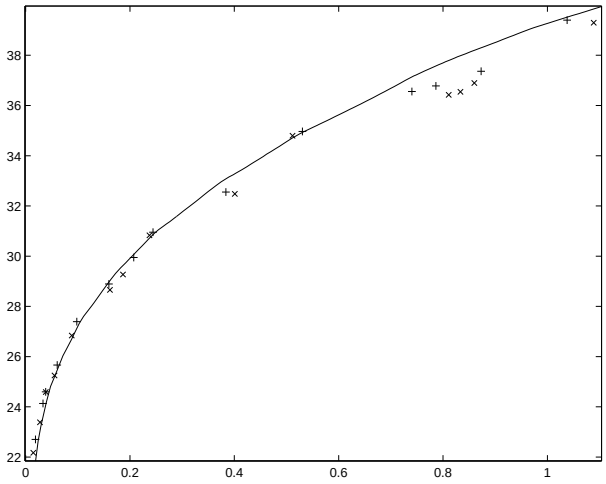

Figure 4.12: R-D plot of Boat image: × OPT(0), + OPT(2), — Jasper

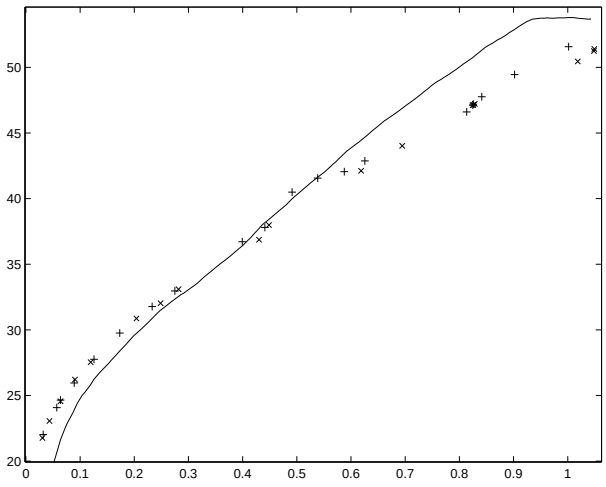

Figure 4.13: R-D plot of Circles image: × OPT(0), + OPT(2), — Jasper

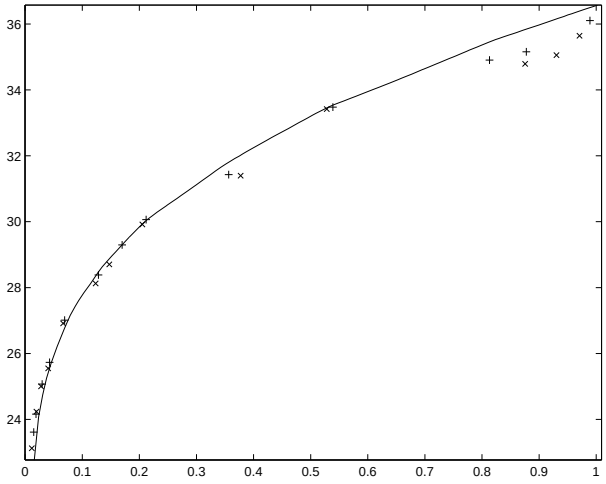

Figure 4.14: R-D plot of Goldhill image: × OPT(0), + OPT(2), — Jasper

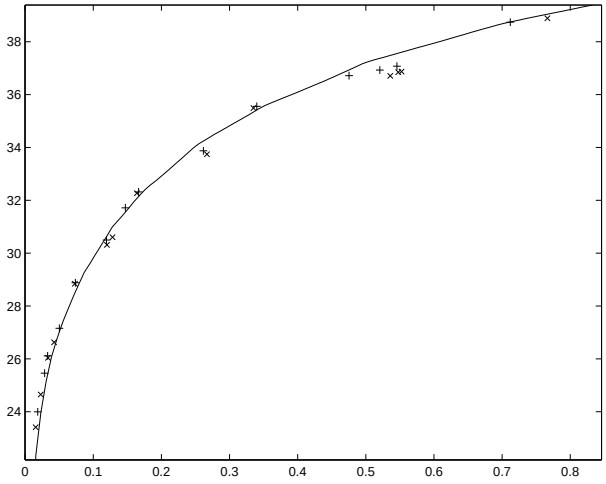

Figure 4.15: R-D plot of Lena image: × OPT(0), + OPT(2), — Jasper

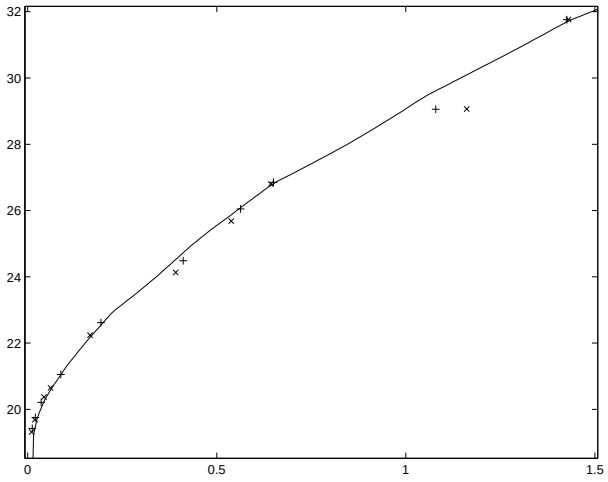

Figure 4.16: R-D plot of Mandrill image: × OPT(0), + OPT(2), — Jasper

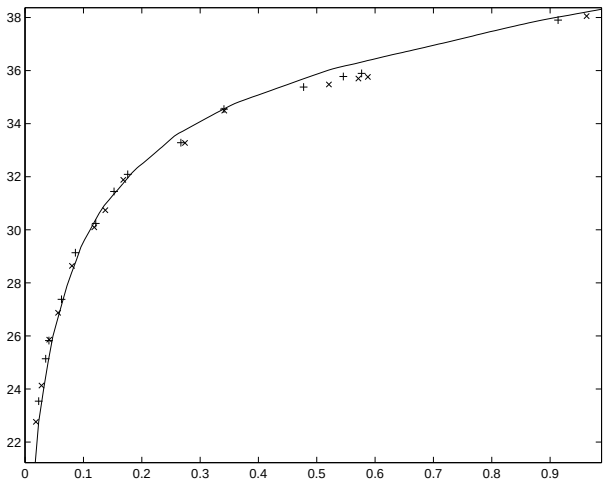

Figure 4.17: R-D plot of Peppers image: × OPT(0), + OPT(2), — Jasper

As one can see, the implemented codec is comparable to Jasper and it is even better for lower rates. The results of image Circles (see fig. 4.13) are very interesting again. The implemented encoder is significantly better than Jasper for very low rates. This example of the difference is depicted in fig. 4.18.

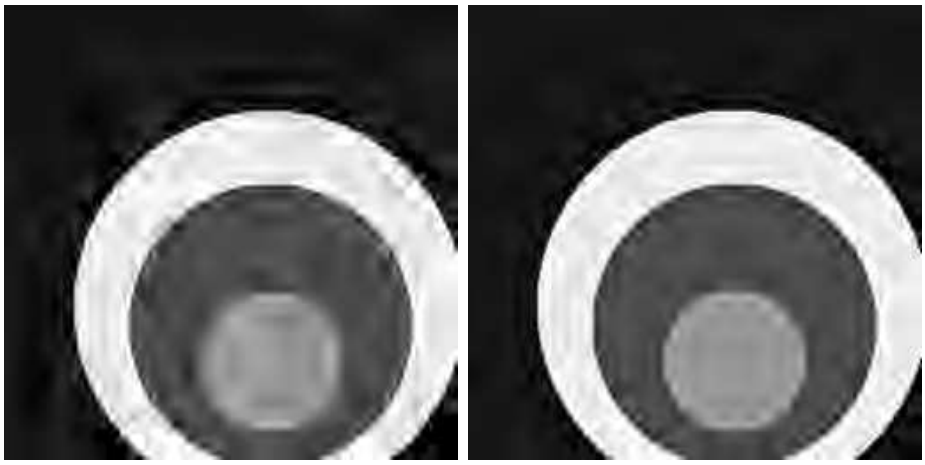

Figure 4.18: Circles images: left Jasper: rate 0.0950, PSNR 24.41; right OPT(0) at `lambda` = 1970: rate 0.0959, PSNR 26.29

4.4 Influence of parameters on optimization

This section discusses the influence of the encoder's parameters on the quality of the resulting image. That means also on the quality of the optimization. The parameters `nl`, `cbp` and `secpar` have only a minor effect, whereas the parameter `lambda` virtually controls the resulting quality.

4.4.1 Influence of lambda

The parameter `lambda` (see section 2.3) determines the truncation point of each codeblock. By changing this parameter, the point at rate-distortion plot will move along the logaritmic-shaped curve which consists of points for all possible values of `lambda`. All images were tested for following values of `lambda`: 10000, 5000, 3000, 2000, 1000, 500, 300, 200, 100, 50, 30, 25, 20 and 10. It is necessary to mention that the same values of `lambda` used in tests

of two different images produce images of different quality. The logaritmic-shaped curves look different for various images too.

4.4.2 Influence of number of decomposition levels

The influence of the parameter `nl` is not very significant. Higher `nl` causes the codeblocks containing the greatest coefficients to be smaller and the coefficients in each of these codeblocks to be more correlated. Therefore they can be optimized more finely. However, the smaller codeblocks also mean smaller statistics for coefficient modelling and higher number of codeblocks in the image. The higher number of codeblocks needs a longer header whose bits are naturally counted into the rate. These two factors limit the upper bound of `nl`. It was discovered that the best values of `nl` are about $\log_2(size) - 3$ where $size$ is either width, or height of the image. Obviously, this simple expression works only for larger images (64×64 and more pixels).

4.4.3 Influence of maximal size of codeblocks

The parameter `cbp` influences the optimization very much because a larger codeblock means larger statistics for coefficient modelling and therefore better performance of the arithmetic encoder. Unfortunately, larger codeblocks also contain coefficients which are not very related to each other and therefore the size of the codeblock is limited. It was revealed that the best values of `cbp` are about $\log_2(size) - 3$ again where $size$ is either width, or height of the image. Obviously, this simple expression works only for larger images.

4.4.4 Influence of secondary parameter

The second optimization procedure was added to the encoder in order to suppress or completely eliminate the disadvantages of the used wavelet transform (see subsection 3.5.6) and also in order to improve on the first optimization stage, which is not always optimal, due to the finite number of points on the rate-distorsion curve, and due to the errors in estimating the real tangent to ideally continuous R-D plot. The improvement of the codec is not so significant for high rates if the second optimization procedure is run. The running time in this implementation is quite high. Generally, the second optimization procedure with `cbp` $= 2$ improves the result about 0.2 dB.

4.5 Conclusions and discussion of the results

The implemented codec illustrates very clearly the importance of optimization for reaching state of the art compression performance with embedded coders. The JPEG-2000 standard was fortunately designed in such a way that similar optimization methods are actually indirectly supported. The optimization method presented in this thesis can be implemented into the JPEG-2000 encoder very easily.

4.5.1 Importance of optimization

An important consequence of the optimizition is the ability to predict the quality of the image from its rate. In case of nonoptimized compression, a higher rate does not automatically imply better quality of the image. However, some very simple optimizations of the codec, like cutting the streams of each codeblock at a certain and constant number of bitplanes before the end, also give good results.

4.5.2 Behaviour at lower and higher rates

The tests, whose results were presented in this chapter, indicate that the implemented encoder produces relatively high quality streams for lower rates. The reason why it is not so successful in higher rates can be in the way the statistical modeller used in this arithmetic coder treats the statistical information, when compared to the finite state model recommended by the standard. Also, the statistical modeller does not use the initial table of symbol probabilities as it is done in the JPEG-2000 standard. Therefore the arithmetic encoder is not able to take an advantage of the coefficient bit modelling when large statisics is already available. This result can also be a matter of the header's length. Other compression codecs probably pay higher fixed cost for the header and metainformation. This can take effect primarily at lower rates. The true reason is very probably a combination of these possible effects.

4.5.3 Phenomenon of Circles image

The Circles image was mentioned in text above twice because the results were unusual. The reason of its unlike behaviour is in both, its smaller size and origin — it is a simple computer generated image with sharp edges and large areas of same grey level.

Firstly, the comparison with SPIHT will be explained. The implemented codec is significantly worse already for rates of about 0.2. This can be explained by the way SPIHT makes use of the correlation between the transformed coefficients in different subbands (by using of tree like structures for zero occurences), while in JPEG-2000 the only correlations which are exploited are those between coefficients in the same subband. The coefficient modelling of the JPEG-2000 based codec is designed mainly for compression of photos. Computer generated images are the weakness of codecs with this kernel. This was proved also by testing photos of the same size. Similar behaviour was not observed. On the other hand, other computer generated images did cause worse results.

The significantly better performance of the implemented codec than Jasper was probably caused by the image's size. Again, other images were tested and they confirmed this result. The reason is presumably in the length of the header. Nevertheless, the used general arithmetic encoder (see section 3.7) can also play some role because it handles a general data better than the arithmetic encoder of the JPEG-2000 standard (see section 1.5).

4.5.4 Summary of implementation

The very first versions of the algorithms were written as Matlab scripts. However, the bit modelling part was very slow in Matlab. It was decided to rewrite it into C language. The acceleration of the implementation was significant, from tens of minutes to less than a second. This stage can be understood as the first stage of the work and it took about four months. The bit modeller conforming the JPEG-2000 standard was implemented during it.

The arithmetic codec was added to the implementation in the second stage which took less than a month. The third stage involved both the first optimization and its testing. This lasted about two months. The second optimization was added in the fourth stage in about a month. One month was reserved for debugging, tuning and adjusting the algorithms.

At the end, the implementation contains over 3300 own code lines in C language and about 100 lines of Matlab scripts. Besides this, about 700 lines of C or Matlab code from other sources were redesigned and modified. The final testing phase, the fifth stage, lasted about two months and over 40 images were altogether heavily tested for many possible combinations of the parameters. Naturally, the tests demanded several hundreds of additional code lines.

The thesis brings the largest contribution in empirical tests of image compression with and without the second optimization showing that the non-orthonormality of the Daubechies filters should not be underestimated. The thesis also points out the differences in coding

between SPIHT (with interband dependency) and JPEG-2000 (codeblock approach) based methods.

4.5.5 Conclusion on the second optimization method

The codeblock partition does not allow in general finding the best point to truncate the stream of the codeblocks in a global sense. The second optimization method overcomes the problem by investigating the truncation points using the rate-distorsion data for the whole image instead of local codeblock data. Although the procedure is time consuming, it is a significant improvement in a comparison with a brute force algorithm. As the performed tests showed, the second optimization method causes a moderate slip along the logarithmic-shaped curve in the R-D plots. This is not, however, the essential problem because the resulting quality of the image is still very high at the resulting rate.

4.5.6 General conclusion and possible improvements

The codec can be improved by several changes. The used arithmetic encoder is apparently very general and the encoder similar to one used in the JPEG-2000 standard would be more appropriate if it should be used mainly for the compression of photos. The general arithmetic encoder is design for arbitrary alphabet, only binary alphabet codec would be a better choice. Its statistical modeller is also very general and does not respect the special case of the compression. The initial probability table can help to improve the results for the photo images although it can worsen the results in a case of the computer generated images. On the other hand, in case of photos, the implemented codec is a better choice than SPIHT because it produces better results and this cannot be caused by a smaller header.

Another improvement of the first optimization procedure can be achieved by additional truncation points cutting the code of the bitplanes. The finer division would make possible to find a truncation point where the slope is closer to `lambda`. However, this would increase a complexity of both the encoder and decoder. Moreover, it would practically disallow the second optimization because more truncation points means a larger state space for searching.

The implemented codec can be used very efficiently for the lossy compression of computer generated images. This can be used, for example, for archiving recognized scene by computer vision systems. In this application, the codec can also successfuly combine these small computer generated images with larger photos of a real scene.

Appendix A

Images used for tests

All test images are grayscale and square-shaped. Their sizes are mostly 512 by 512 pixels.

1. **Barbara** is a 512 by 512 pixels large image of a woman sitting on the floor in the middle of a room. The picture contains objects with fine texture with sharp edges in various directions like scarf, trousers or a wicker armchair.

Figure A.1: Barbara

2. **Boat** is a 512 by 512 pixels large image of a fishing vessel lying on a bottom of harbour during a flow. The picture contains sharp edges of masts and ropes.

Figure A.2: Boat

3. **Circles** is a 256 by 256 pixels large computer generated image of three eccentrically placed circles of different grades. The image contains several sharp edges.

Figure A.3: Circles

4. **Goldhill** is a 512 by 512 pixels large image of a street on a steep hill with couple of houses and landscape scenery in the background. The picture contains many different textures, for example walls, windows and distant trees.

Figure A.4: Goldhill

5. **Lena** is a 512 by 512 pixels large image of woman's face. The picture contains many small details like hair or fine feathers but also quite huge areas without edges.

Figure A.5: Lena

6. **Mandrill** is a 512 by 512 pixels large image of Mandrill Baboon's face. The image contains enormous amount of textures and edges.

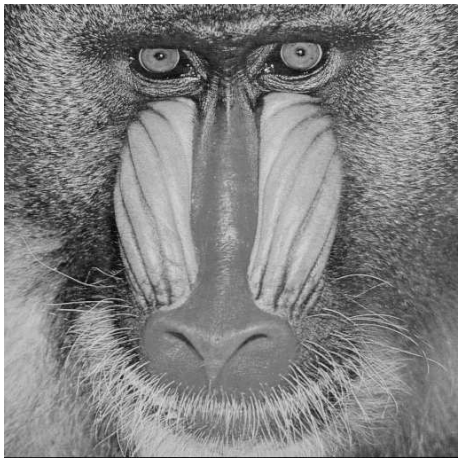

Figure A.6: Mandrill

7. **Peppers** is a 512 by 512 pixels large image of a collection of peppers. The image contains mainly smooth transitions and a few easily distinguishable edges.

Figure A.7: Peppers

Appendix B

Usage of other encoders

B.1 SPIHT usage

Set Partitioning in Hierarchical Trees [10] is a wavelet-based image compression method. The SPIHT software is written in C++ programming language. Its source code is not publicly available but the compiled binaries are, at http://www.cipr.rpi.edu/research/SPIHT/. For test purposes, the version 8.01 was used. The command which executed the encoder during testing was

codetree `origfile` `spihtfile` *height width bytesPerPixel rate*

where `origfile` represents the original file in a raw format, `spihtfile` is the resulting file, *height* and *width* are the dimensions of the image, *bytesPerPixel* expresses the number of bytes per pixel and finally *rate* denotes the target rate.

B.2 Jasper usage

Jasper [18] is an implementation of the codec specified in the emerging JPEG-2000 Part-1 standard. JasPer software is written in C programming language, therefore it is very fast. Its source code is available at http://www.ece.uvic.ca/~mdadams/jasper/. For test purposes, the version 1.701.0 was used. The command which executed the encoder during testing was

jasper -f `bmpfile` -F `jp2file` -T jp2 -O mode=real -O rate=*rate*

where *rate* was chosen between 0 and 1. Parameters from left mean: (-f) use a source image of bmp format from the file `bmpfile`, (-F) output encoded image into the file `jp2file`, (-T) output format is jp2, (-O mode) compute everything by floating point operations and (-O rate) create a file which will have a rate *rate*.

Index

Bibliography

[1] *Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*, ISO/IEC IS 10918-1, ITU-T Recommendation T.81, 1994.

[2] *Lossless and near-lossless compression of continuous-tone images — baseline*, ISO/IEC 14495-1, ITU-T Recommendation T.87, 2000.

[3] *Information technology — JPEG 2000 image coding system: Core coding system*, ISO/IEC 15444-1, ITU-T Recommendation T.800, 2000

[4] M. D. Adams: *The JPEG-2000 Still Image Compression Standard*, ISO/IEC JTC 1/SC 29/WG 1 N 2412, Dec. 2002.

[5] W. Sweldens: *The lifting scheme: construction of second generation wavelets*, SIAM J. Math. Anal., vol.29, no. 2, pp 511-546, 1997

[6] C. Christopoulus, J. Askelof and M. Larsson: *Efficient methods for encoding regions of interest in the upcoming JPEG2000 still image coding standard*, IEEE Signal Processing letters, vol. 7, pp. 247–249, Sept. 2000

[7] D. Singer, R. Clark and D. Lee: *RFC 3745 - MIME Type Registrations for JPEG 2000 (ISO/IEC 15444)*, IETF repository, April 2004.

[8] D. Taubman: *High Performance Scalable Image Compression with EBCOT*, IEEE Transactions on Image Processing, vol. 9, no. 7, pp. 1158–1170, 2000

[9] J. M. Shapiro: *An embedded hierarchical image coder using zerotrees of wavelet coefficients*, IEEE Data Compression Conference, Snowbird, UT, USA, 1993, pp. 214–223

[10] A. Said and W. Pearlman, *A new, fast and efficient image codec based on set partitioning in hierarchical trees*, IEEE Trans. Circuits and Systems for Video Technology, vol. 6, pp. 243–250, June 1996

[11] S. Mallat, *A theory for multiresolution signal decomposition: The wavelet representation*, IEEE Trans. Pattern Anal. Machine Intell., vol. 11, pp. 674–693, July 1989

[12] *EBCOT: Embedded Block Coding with Optimal truncation*, ISO/IEC JTC1/SC29/WG1 N1020R, Oct. 1998

[13] *Generalized Lagrange multiplier method for solving problems of optimum allocation of resources*, Oper. Res., vol. 11, pp. 399–417, 1963

[14] D. Taubman and A. Zakhor, *Multi-rate 3-D subband coding of video*, IEEE Trans. Image Processing, vol. 3, pp. 572–588, Sept. 1998

[15] M. Antonini, M. Barlaud, P. Mathieu and I. Daubechies, *Image Coding Using Wavelet Transform*, IEEE Trans. Image Processing, vol. 1, pp. 205–220, Apr. 1992

[16] J. Rissanen: *Generalized Kraft Inequality and Arithmetic Coding*, IBM Journal of Reasearch and Development, vol. 20, no. 3, pp. 198–203, 1976.

[17] I. Tăbuş, G. Korodi and J. Rissanen: *DNA sequence compression using the normalized maximum likelihood model for discrete regression*, Proc. IEEE Data Compression Conference (DCC 2003), pp. 253–262, 2003.

[18] M. D. Adams and R. K. Ward, *JasPer: A Portable Flexible Open-Source Software Tool Kit for Image Coding/Processing*, in Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, PQ, Canada, May 2004.